MSX

MARGARET NORMAN

MADE SIMPLE COMPUTER BOOKS

MSX Made Simple

IN THE SAME SERIES

BASIC
Computer Electronics
Computer Programming
Computer Programming Languages in Practice
Computer Typing
Data Processing
Systems Analysis
Word Processing for the Professions

MSX Made Simple

Made Simple Computerbooks

Margaret Norman

MADE SIMPLE
COMPUTER
BOOKS

HEINEMANN : London

Printed and bound in Great Britain
by Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk
for the publishers William Heinemann Ltd,
10 Upper Grosvenor Street, London W1X 9PA

British Library Cataloguing in Publication Data
Norman, Margaret
MSX made simple. – (Made simple books,
ISSN 0265-0541)
1. MSX microcomputers – Programming
I. Title II. Series
005.2'6 QA76.8.M8

ISBN 0 434 98406 X

Contents

Introduction

This book is aimed at anyone who has the use of an MSX computer. It is intended to teach MSX BASIC from first principles, and to provide a brief introduction to machine code. The material has been arranged to make the transition from BASIC to machine code as simple as possible – hence the early introduction of binary and hexadecimal numbers.

The chapters should be worked through in order, as each chapter uses concepts introduced in earlier chapters. Numerous examples are included throughout the book, and it is intended that the reader should try out as many of these as possible. Each example program is followed by a line-by-line explanation, to reinforce understanding of the statements used.

Where appropriate, answers to the exercises are given at the back of the book. There are numerous possible correct answers to some of the programming exercises; any program which performs the required function can be regarded as acceptable.

1

What is MSX?

1.1 WHY MSX?

The last few years have seen a boom in the production and sales of home computers – that is, computers which are small, cheap and simple enough to be owned and used by ordinary individuals and families, for education, entertainment and practical applications such as keeping accounts and word processing. There are many different models on the market, and they are very widely available, from chain stores as well as specialist shops.

Look in any shop which sells computers and you will see not only a range of different computers, but also a wide selection of *peripheral devices* – the monitors, cassette players, disk drives, printers and other items which can be plugged into the computer to make up a complete system – and a large amount of commercial *software* – the programs which enable the computer to perform different tasks. Almost all of these items are marked with the make and model of computer for which they are intended. They are *machine specific*, i.e. they can be used only with one particular computer. It is a strange state of affairs; it is as if every different model of record player had to have its own special speakers and records, which could not be used with any other model.

When a new computer is launched, the manufacturers normally try to ensure that there will be some peripherals and software available for it immediately. However, it is not practicable, in such a rapidly developing field, for manufacturers to delay launching a new machine until all the possible peripherals have reached the production stage and all the software which might be required has been written. The initial range is usually small, with further items promised for some time in the future. Many computer manufacturers produce only a limited range themselves, and rely on the support of independent peripheral manufacturers and software houses to make additional items available. If the

new machine proves popular then this support will almost certainly be forthcoming. However, most software houses understandably prefer to write their programs for computers which have already sold in large numbers, to maximize their potential market, and most purchasers of computers are understandably reluctant to buy new machines which, though they have better specifications than older models, may never be able to realize their full potential because of a lack of software support. It is thus very difficult for a new, non-standard machine to become established, and several very promising machines have failed in the past few years, leaving their owners with expensive white elephants on their hands.

The advantages of standardization are very apparent. If only the same programs and peripherals could be used with all home computers, the risks inherent in producing a new machine, and in buying a machine new to the market, would be greatly reduced. Retailers would no longer have to stock numerous different versions of each product, and could instead carry a much wider range of products. Programmers would no longer have to rewrite their programs to run on different machines, but could concentrate on improving existing programs and producing new ones.

The *MSX standard* is the first serious attempt to produce a widely accepted standard for home computers. MSX, named after the MicroSoft eXtended BASIC with which the machines are supplied, sets down a minimum standard for home computers; all the computers conforming to this standard can use any of the MSX peripherals and software on the market. Initially supported by several major Japanese electronics companies, MSX is now gaining ground all round the world.

Manufacturers are free to add extra facilities to MSX machines, so the fact that they all conform to the same standard does not mean that they are all identical. There is already a wide range of models, from low-priced starter computers to sophisticated models with extra memory capacity, built-in software and various special facilities. Acceptance of the standard does not mean that consumers are faced with a limited choice; it does mean, though, that they can rest secure in the knowledge that all the add-ons and software they might need are readily available now, and will continue to be available for years to come.

1.2 THE MSX STANDARD: THE PROCESSORS AND MEMORY

The most important part of a computer is the silicon chip known as the *central processing unit*, or *CPU* for short. This chip performs the computation and manipulation of data. There are several different types available. The word length of the CPU, the number of binary digits (bits) it can handle in parallel, determines the accuracy with which the computer can perform arithmetic operations, and the speed with which the processing is carried out. Most home computers have 8 bit CPUs; the one chosen for MSX computers is the 8 bit Zilog Z80A or equivalent, a very popular and well-proven processor.

The choice of CPU determines the *machine code* with which the computer has to be programmed. Each processor has its own machine-code instruction set. However, writing programs in machine code is difficult and time consuming, so home computers are usually supplied with a special program known as a *BASIC interpreter*. This will translate, into machine code, programs written in BASIC, a *high-level language* which is easier to understand and use. There are numerous different high-level computer languages, but BASIC, which stands for Beginners All-purpose Symbolic Instruction Code, predominates in the home computer field. The BASIC interpreter is supplied on special memory chips, the contents of which are permanently fixed. This type of memory is known as *read-only memory*, or *ROM*. Each memory location can contain 8 binary digits, or 1 *byte*. One *kilobyte* (1K) of memory is 1024 bytes. MSX computers have at least 32K of ROM. (Some models have additional ROM which contains built-in software.)

The contents of *random access memory*, or *RAM*, are not fixed, and are lost whenever the computer is switched off. This type of memory forms the computer's work space; it is used to hold BASIC or machine-code programs, program data, and program and system variables. The larger this work space is, the better; computers with only a little RAM can handle only short programs and small amounts of data. MSX computers have a minimum of 8K of user RAM; most models have 32K or 64K.

A special section of RAM, the *video RAM* or *VRAM*, is used to store information about the screen display. MSX computers have 16K of VRAM. Some non-MSX computers store screen information in the main user RAM instead – meaning, of course, that they have less space available for storing programs and data.

A special *video display processor* (VDP) is used to control the VRAM.

The VDP used in MSX computers, the TMS 9929A or equivalent, provides four different screen modes, sixteen colours and up to 32 sprites.

There are two different text modes, in which a range of 256 different characters, including special graphics characters, can be used. These modes enable you to have up to 40 columns of text on the screen (i.e. 40 characters per screen line). There are some applications, e.g. word processing, where it would be desirable to have more columns of text than this – 80 columns are provided on most business computers – but the quality of the picture produced on an ordinary television set is not good enough for 80 column text to be legible. However, if you want to use your computer for word processing, it is possible to buy a special '80 column card' and a high-resolution monitor to produce clear 80 column text.

The graphics modes offer two different *resolutions*. The resolution of the screen is the number of locations which can be coloured individually; the higher the resolution, the sharper the picture. There is a *low-resolution* multicolour mode where the screen is divided into 64 columns of 48 little squares (a resolution of 64×48), each of which can be assigned any of the sixteen available colours, and a *high-resolution* mode where there are 256×192 *pixels*, or little dots, on the screen and detailed pictures can be produced.

A third processor, the *programmable sound generator* (PSG), enables the computer to produce sound. The processor chosen, the General Instruments AY-3-8910 or equivalent, can generate three channels of periodic (musical) sound and one channel of white noise, with a range of eight octaves. It can be programmed to play three-part music, and to produce a wide range of different sound effects.

1.3 INPUT AND OUTPUT: THE KEYBOARD AND PORTS

Computers are not (yet) intelligent; they can do nothing unless they are given instructions, in the form of different commands or a program of commands to be executed on demand, and data. There are several different ways to input instructions and data; the most obvious input device is the keyboard.

Computer keyboards vary considerably in layout and construction. Some do not have proper moving keys, just pressure-sensitive pads.

Some have all manner of strange symbols printed on the keys. MSX computers have keyboards which look much like standard typewriter keyboards, with the addition of a few special keys; they have a minimum of 73 keys altogether. The alphanumeric keys are laid out using the QWERTY system, which will be familiar to any typist. There is a space bar and a shift key, which operate in the same way as their equivalents on a typewriter. There are also some special keys, labelled CAPS, CODE and GRAPH, which are used to access alternative sets of characters. Another group of special keys is used for editing programs and text: CLS/HOME, INS and DEL. A row of five *function keys* is provided, normally along the top of the keyboard; these can be programmed to perform various functions. There is a cluster of *cursor keys*, used to move the cursor (which indicates the print position) around the screen, and also used in many games and other programs. Finally, there are the RETURN, CTRL, TAB, ESC, SELECT and STOP keys, the functions of which will be explained later.

The MSX standard does not specify any other built-in input devices, or any built-in output devices. Instead, the computers are fitted with a range of *ports* (sockets), into which you can plug the devices of your choice. This system provides considerable flexibility at a comparatively low cost.

The most essential form of output is a *visual display*, provided by either an ordinary television set or a purpose-built monitor. The visual display produced by the computer starts off in the form of separate red, green, blue and synchronizing signals – *RGB* output. Some MSX computers have a RGB output port, to enable a special RGB monitor to be connected to the computer, but this is an optional port and most models do not have it. The signals can be combined to produce a *composite* signal: a port is provided to enable connection to a monitor designed to accept composite input. The composite signal is vision only, not sound, so a separate audio port is also provided. The composite signal can be passed through an *RF modulator* to produce the RF signal required by an ordinary television set; this signal incorporates a sound signal. An RF port is provided as standard.

A *cassette interface* is also provided as standard, to enable the computer to be connected to a standard cassette player. Some non-MSX computers have special cassette interfaces and so need special cassette recorders, but MSX computers have been designed to work with ordinary cassette recorders (though there are some special recorders on the market for them). The cassette interface allows a data transfer rate of 1200 or 2400 *baud* (bits per second).

The *Centronics parallel interface* allows the computer to be connected

to a Centronics *printer*. Special MSX printers are available, but you can also use non-MSX Centronics printers, though these may not be able to reproduce the graphics characters.

Most MSX computers have two *joystick* ports, though the standard specifies only one. They are standard D-shaped ports, which will accept most popular makes of joystick, though it is best to buy special MSX joysticks: these have two fire buttons, whereas many other joysticks have only one.

There are also cartridge and expansion slots, which allow the use of ROM cartridges and a wide range of other peripherals, including disk drives, light pens and modems.

1.4 MSX BASIC

Almost all home computers are supplied with a BASIC interpreter in ROM. However, there is no universally accepted form of BASIC; there are nearly as many different dialects as there are different computers. Some versions are extremely good and provide a wide range of different commands, while others are comparatively limited in their scope. All BASICs include a range of mathematical and data-handling commands; the main variations are in the fields of graphics and sound.

The dialect of BASIC used on MSX computers is *Microsoft Extended BASIC* (MSX BASIC). This is one of the most comprehensive BASICs available, with a full range of graphics and sound commands, as well as all the standard operators and functions and special commands to facilitate error trapping and the use of interrupts. Even newcomers to computing should be able to write quite complex programs in this language in a comparatively short time. Entering programs is simple too, because a full screen editor is provided to enable mistakes to be corrected easily.

The quality of the BASIC interpreter affects not only the ease with which programs can be written, but also the speed at which they will run. Some high-level computer languages are *compiled*, i.e. the whole high-level language program is translated into a machine-code program which is then *executed*. The BASIC interpreter does not compile programs, however; instead, when the program is run, it looks up each BASIC command in turn in a table which refers it to a corresponding machine-code routine. This is a time-consuming business, and explains why BASIC programs run much more slowly than their machine-code equivalents. The greater the number of BASIC commands which are

needed to perform a particular function, the slower the execution of the program will be.

There is a lot to be said for having a comparatively standard form of BASIC, if not identical in all respects to the dialects used by many other computers, is at least not very dissimilar to them. It is quite easy to convert a BASIC program written for one computer to run on another if only a few of the commands differ. The most widely used form of BASIC is *Microsoft BASIC*, which is used in slightly different forms on a wide range of computers. Many computer books and magazines contain programs written in Microsoft BASIC; it is also used for the demonstration programs in many printer manuals and similar publications. MSX BASIC is an extended form of standard Microsoft BASIC, so most Microsoft BASIC programs should require little, if any, alteration to run on MSX computers.

1.5 COMMERCIAL SOFTWARE

A computer without software is like a library without books; the computer needs programs to tell it what to do and how to do it, and if you are to use your computer to the full, you will need a good range of different programs for it. Some of these you may be able to write for yourself, but as writing programs is a very time-consuming process, you will inevitably want to buy some.

Commercial software falls into four main categories. First there is business software: word-processing programs, databases, spreadsheets, accounts packages and so on. Then there are *utilities*: programs designed to help you make the best possible use of the computer, by making it easier for you to use the graphics and sound facilities or to enter and edit machine-code programs, or by adding extra commands to the built-in BASIC. Next come educational programs, for children and for adults: programs to improve your mathematics, to help you to learn a foreign language, to test your general knowledge, in fact to teach you just about anything. Last, but by no means least, there are the games: arcade games, conversions of board games, every sort of game you could think of and a few you could never have imagined.

The number of programs available for MSX computers is already well into three figures. Some of the programs on the market are very good; some are hopelessly bad. The most expensive ones are by no means invariably the best, so it is well worth reading program reviews in magazines, or if possible trying the programs out for yourself, before

deciding what to buy.

Most programs are supplied on cassette, and come complete with any necessary loading instructions. Some are supplied instead on ROM cartridges, which can be plugged into the cartridge port on the computer. You should always switch the computer off before inserting or removing a cartridge – if you do this while the computer is on, you could cause considerable damage. Cartridges are very convenient as the programs on them do not have to be loaded into RAM: the cartridge program is instantly available as soon as the computer is switched on. However, they are rather more expensive than cassettes. For owners of disk drives, a selection of programs – mainly business programs – are available on floppy disks. MSX disk drives are not standardized, so you must ensure that the programs you buy on disk are compatible with your drive.

2

Choosing an MSX Computer

2.1 HOW MSX COMPUTERS DIFFER

Although MSX computers all conform to the same minimum standard, this does not mean that they are all the same. There are considerable variations in the facilities they offer, the standard of construction, the appearance and, of course, the price. So which should you choose?

All the machines currently available have their strengths as well as their weaknesses; any machine which was all bad would very soon disappear from the market. Which machine is best for you will depend on your own priorities and on the size of your budget.

The most important point to look for is the amount of RAM which is provided. Any MSX program will run on any MSX machine with sufficient RAM, but it is obviously not possible for a program which takes 64K of RAM to run on a machine which has only 8K available. You should buy a machine with a small amount of RAM only if you are sure that you will not need any more for the programs you intend using. It is possible to buy expansion packs which will increase the amount of RAM available, but it's better to have it built in.

If you intend using a monitor, then you must ensure that the computer has the right ports for your model. A machine which offers only RF and composite outputs is not much use if you have an RGB monitor.

Look, too, at the number of cartridge ports provided, the number of joystick ports, and any other special ports. Look also at the position of these: would you find it more convenient to drop cartridges into a slot on top of the machine, or to have the port at the back? Would joystick ports on the front edge be more convenient than ones at the side?

A reset button is a useful extra: it enables you to stop a program and re-initialize the computer without switching the power off. Indicator lights to show when the power is on, and when the <CAPS>,

<GRAPH> or <CODE> keys are depressed, are also useful; check that these are well placed and clearly visible.

Look carefully at the keyboard, particularly if you intend using the computer for word processing. Some are more responsive, and have a better 'feel' to them, than others. Check that the key markings are clear and durable. The layout of the main partof the keyboard is standard, but there are considerable variations in the size and shape of the function and cursor keys. Some models have a separate numeric keypad, which can be very useful if you have a lot of numeric data to enter. On some computers, different colours are used for different groups of keys, which can be a help if your typing is indifferent.

The overall quality of construction is important, though less easy to assess. Does the machine look sturdy, or is its casing rather flimsy? Does it become unacceptably hot when in use for a long period? Are there grilles to help keep it cool, and if so, are they well designed and well placed?

Most machines are guaranteed for twelve months, but some have longer guarantees, which are obviously valuable. Try to discover how easy it will be to get repairs carried out if something goes wrong. Can the supplier do minor repairs, or will the machine have to be returned to the manufacturer?

The standard of the manuals supplied with computers varies considerably. There are plenty of books (like this one) available to help you if your manuals are less than adequate, but do check to see if they are clear and comprehensive.

Some computers come complete with software; this may be built in, or supplied in cassette or cartridge form. It may be very good, or completely useless. Remember, though, that you need not judge MSX computers by the quality of the manufacturer's software; you are free to use software produced for other MSX machines.

2.2 A SELECTION OF MACHINES

Space in this book does not permit full reviews of all the different MSX computers on the market here. These are just brief descriptions of some of the first models available in the UK.

Sanyo MPC-100

This is the computer which was used for writing this book, and is at the upper end of the price range. It is an elegant looking machine, with black and silver casing and a rather better than average keyboard.

The facilities provided are fairly standard. It has 64K RAM, two cartridge ports, two joystick ports and a reset button as well as the usual Centronics printer port, cassette port, RF and composite outputs. The joystick ports and reset button are on the side of the machine, and all the other ports are on the back.

The cursor keys are nicely designed, and the function keys are large and well placed. My only complaint about the keyboard is the position of the STOP key, which is rather too far from the CTRL key for convenience.

A brief operating guide and a rather more substantial programming manual are provided. The latter gives a good summary of all the BASIC commands, arranged in alphabetical order for ease of reference, but does not provide an adequate BASIC tutorial for beginners. The English language used is rather strange at times, presumably because it has been translated from Japanese.

The package includes two cassettes, one containing two games and the other offering a guide to the system and typing tutor. Both are of very poor quality.

If you want a standard 64K machine, this one is worth a close look.

Spectravideo SVI-728

This is a rather cheaper computer, from a Hong Kong based company. The same company produces a large range of peripherals, including a disk drive, 80 column card and modem. They answered my request for information with impressive speed.

The computer looks noticeably different from most of the other MSX computers on the market, mainly because it has a separate numeric keypad. The cursor keys are arranged along the top of this pad, and so are less easy to use than those on most other machines. There is a detachable power cable complete with transformer; the computer is therefore rather smaller than those which have transformers built in.

The facilities provided are adequate, but not outstanding: 64K RAM, an expansion bus (with a non-standard shape cutout which may make the peripherals rather awkward to fit on other MSX micros), a cartridge port, two joystick ports and the other standard interfaces.

The manual is better for beginners than the Sanyo manuals, but less comprehensive in its coverage of BASIC commands. No software is supplied.

Spectravideo are clearly aiming their products at business users rather than games players. This computer appears to be good value for what it offers.

Toshiba HX-10

The most widely available MSX micro in the UK, the Toshiba has been marketed more aggressively than any other, with special offers of good-quality games software, extended guarantees and price cuts. It is currently very cheap compared with the other Japanese machines available here.

The styling is two-tone grey, with the cursor, <STOP> and <GRAPH> keys picked out in bright colours, which gives it a rather garish look. The keyboard design is fairly standard, but the cursor keys are smaller than average, though well arranged for games playing. The keyboard is not particularly good for touch typing, as it is rather flat.

The HX-10 lacks a reset button, but otherwise has the same specification as the Sanyo. There are two manuals, covering the same ground as the Sanyo ones but doing it rather better.

Competitive pricing makes this machine a good buy if you do not intended doing too much typing on it.

Yamaha CX-5M

The Yamaha is very different – not so much a home computer as a music synthesizer that happens to incorporate an MSX computer. It comes complete with a music keyboard, which has a non-standard interface and so cannot be used with other MSX machines. A MIDI interface, the industry standard to which a range of other synthesizers, drum machines and other instruments can be connected, is also provided.

Yamaha have produced a range of software to go with this unique computer, including a music composer which allows you to write music by inputting notes from the computer keyboard or the music keyboard, a voicing program and a music macro which allows you to control the FM voice generator from within BASIC programs. There are eight music channels to play with, so the scope of the system is very considerable.

Where the computer part is concerned, it is a standard 32K MSX computer with all the usual interfaces and an adequate, if unexciting, keyboard. It functions perfectly well just as a computer, but Yamaha clearly do not expect anyone to buy it primarily for mainstream computing, and are currently marketing it through music shops rather than computer shops.

When regarded purely as a computer, the Yamaha is very expensive – twice the price of other MSX micros – but if you compare it instead with other synthesizers, then it appears quite cheap. And of course, if you want a synthesizer, then the fact that you get a computer thrown in with this one is a great bonus.

2.3 SETTING UP THE SYSTEM

Setting up a basic system comprising the computer itself, a television and a cassette recorder is a simple matter and is covered in detail in most manuals.

The three pieces of equipment must all be connected to the mains electricity supply. If you do not have three separate power points available, do not worry; a four-gang trailing socket, which can be purchased quite cheaply from any electrical shop, will enable you to plug everything into a single power point, and will leave you with a spare socket for any additional equipment you wish to use.

Do not switch on the mains supply until the television and cassette recorder have both been connected to the computer. The leads required are normally supplied with the computer. If you do not have them, or if you require additional leads, to use equipment such as a monitor or a printer, ask your dealer for advice.

The RF lead is used to connect the RF output terminal on the computer to the antenna terminal on the television. If the television is not to be used solely with the computer, you can purchase a two-way connector which will allow you to plug both the computer lead and the antenna lead into the antenna terminal at the same time. This will prevent the wear and tear on the leads which will inevitably result if you keep swapping them over.

The cassette recorder lead should have a round eight-pin DIN socket at one end, which fits the cassette port on the computer, and three jack plugs at the other end. The cassette recorder should have two sockets, labelled EAR and MIC(rophone), into which the two larger jack plugs will fit. Check in your manual to see which is which; they are normally

colour coded for easy identification. If you plug them in the wrong way round you are very unlikely to do any damage to the computer or cassette recorder, but your attempts to load and save programs will be totally unsuccessful!

The third, slightly smaller, jack plug fits the REM(ote) socket on the cassette recorder. When this is connected, the computer can control the motor in the cassette recorder. If your cassette recorder does not have a REM socket you can simply ignore this plug and switch the recorder on and off manually.

When you are satisfied that the equipment has been connected up satisfactorily, switch on first the peripherals, then the computer. Do not worry if you do not immediately get a picture on the television; this will probably be because you have not tuned it into the right channel. When you have tuned it in, the copyright message shown in Figure 1 should appear. The print should be white, on a dark blue background (unless, of course, you are using a black and white television, when you will see white print on a black background). Adjust the television controls until the picture is clear and steady.

```
MSX BASIC version 1.0
Copyright 1983 by Microdeal
xxxxx Bytes free
Ok


color     auto     goto     list     run
```

Figure 1 *The copyright message*

3

Entering, Saving and Loading Programs

3.1 USING THE KEYBOARD

When the computer is first switched on, a small white square should appear on the screen below the copyright message. This is the *cursor*. Type a letter, and it should appear on the screen at the cursor position; the cursor will automatically move to the right as you type. When you reach the end of a line, the cursor will move to the start of the next line. You can type whatever you want, without damaging the computer in any way.

Whenever a word is shown in broken brackets < >, this indicates that a special key is being referred to. Do not type the word inside the brackets on the keyboard, just press the appropriate key.

If you press <**RETURN**>, to the right of the top two rows of letter keys (marked with a large arrow on some machines), the computer will probably respond with an *error message*: 'Syntax error'. This is because the computer has tried, and failed, to identify what you have typed as a valid BASIC command. If what you have typed *is* a valid command, the computer will either respond to it or, possibly, produce a different error message. If the cursor disappears, or anything else inexplicable happens, press the reset button (if there is one) or switch the computer off then on again to return to the copyright message.

Playing around with the keyboard will reveal several interesting features. If you hold a key down, after a short delay (to ensure you have held it down on purpose) it will *auto-repeat*, i.e. the appropriate letter or symbol will be printed again and again until the key is released. When you type normally, you should obtain lower-case letters, and the numbers and lower symbols on the symbol keys. If you hold down one of the <**SHIFT**> keys (at either end of the bottom row of letter keys) while typing, you should obtain capital letters, and the upper symbols. If you press <**CAPS**> – do not hold it down, it is a *toggle* key and

remains operative until pressed again – you should get capital letters, with the numbers and lower symbols. (A small light is often provided to indicate when CAPS mode is operative.) If you hold down <**GRAPH**> while pressing the other keys, a range of graphics characters will appear. <SHIFT> and <GRAPH> together will produce a different range of graphics characters. <**CODE**>, and <CODE> and <SHIFT> together, produce yet more characters. Figure 2 shows which characters are produced by various combinations of keys.

The *cursor keys* move the cursor around the screen, without altering what you have typed. Try to produce a picture, placing graphics characters on the screen in positions selected using the cursor keys. If you want to clear the screen before you start, <SHIFT> and <**CLS/HOME**> will do this for you.

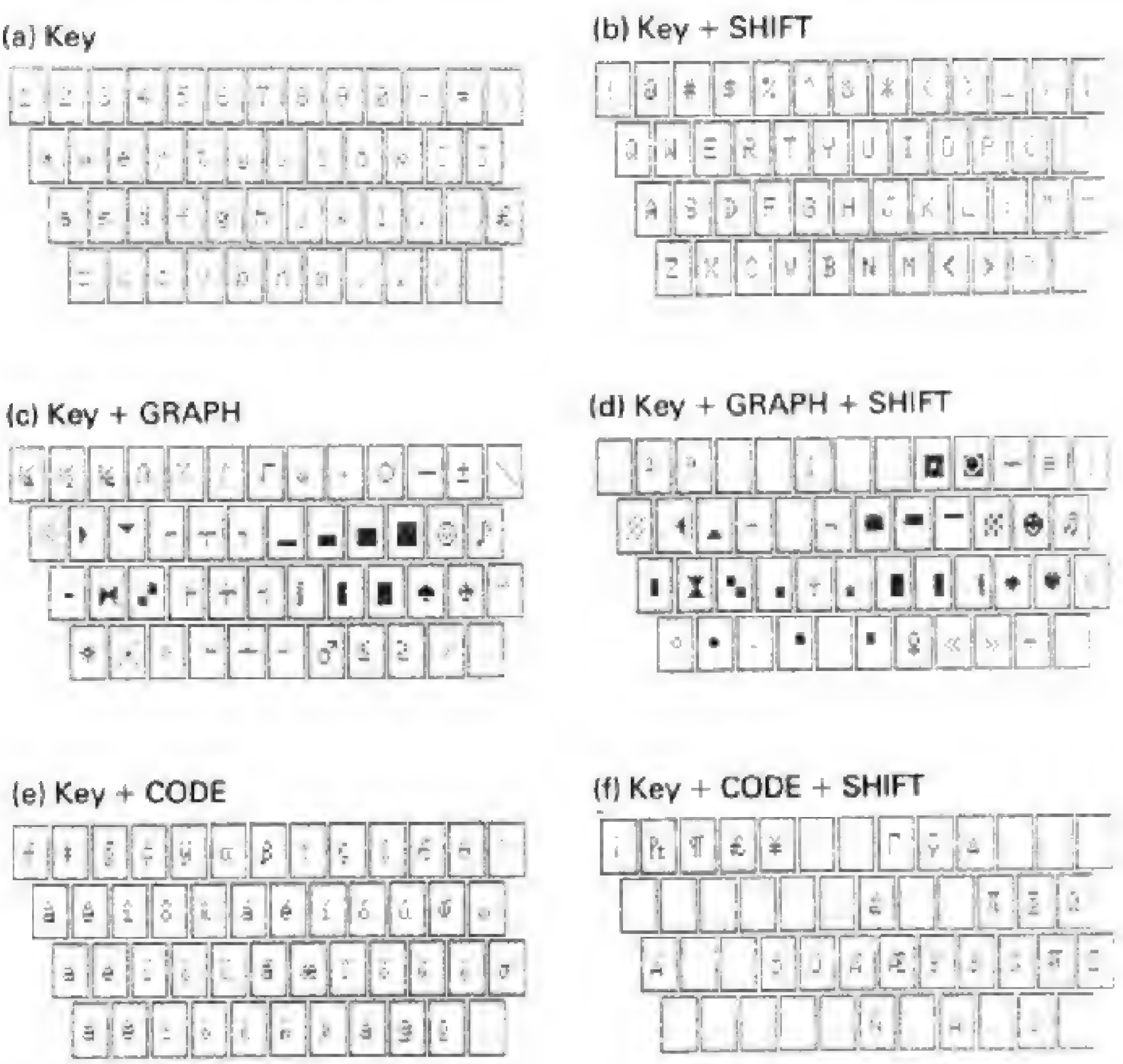


Figure 2 *Keys and characters*

Further experimentation will reveal that the backspace key <**BS**> deletes the character immediately behind the cursor; the delete key <**DEL**> deletes the character in front of the cursor. Both of these also move all characters to the right of the cursor one space back. There are three other ways to delete unwanted characters: the character at the cursor position can be deleted without moving any other characters by simply typing a space; all characters to the right of the cursor can be deleted by pressing the control key <**CTRL**> and **E**; and all characters on the same line as the cursor can be deleted with <CTRL> and **U**.

The insert key <**INS**> puts you into insert mode; the shape of the cursor changes, and any characters you type in this mode are inserted into the text at the cursor position. <INS>, like <CAPS>, is a toggle key, so pressing it again cancels insert mode. <CTRL> and **R** perform the same function as <INS>.

Pressing <SHIFT> and <CLS/HOME> together, as we have seen, clears the screen. UnSHIFTed <CLS/HOME> moves the cursor to the top left-hand corner of the screen, without clearing it. <TAB> moves the cursor to the next tab position; the tab position are at eight-column intervals across the screen. There is a 'dead' key, used for adding foreign accents to letters, which does not move the cursor at all. <ESC> and <SELECT> are not used; nothing will happen if you press these. The function of <STOP> will be explained later.

The words 'color', 'auto', 'goto', 'list' and 'run' are normally displayed at the bottom of the screen. You may have noticed that these change when <SHIFT> is pressed, to 'color', 'cload"', 'cont', 'list.' and 'run'. These are the functions assigned to the five *function keys* at the top of the keyboard. Press a function key, and the appropriate command should appear. Note that <F6> (<SHIFT> and <F1>) produces not just 'color', but 'color 15,4,4' – a command which restores the normal dark blue and white colour scheme, if the colours have been changed by a BASIC command or program.

The <CTRL> key functions are summarized in Appendix E.

Exercise 3.1

Type on the screen:

Ths iss a keyborrd exercise!!!!!!

Now alter this to read:

This is a keyboard exercise.

using (a) <INST>, <DEL> and <BS>
(b) <CTRL> and the appropriate letter keys.

Exercise 3.2
Type on the screen:

F1	color	F6	color 15,4,4
F2	auto	F7	cload"
F3	goto	F8	cont
F4	list	F9	list.
F5	run	F10	run

(*Hint*: use <TAB>. Watch out for error messages! Do not use <F10> for the final 'run', or the screen will clear and you will have to start all over again.)

3.2 ENTERING AND EDITING PROGRAMS

A BASIC program consists of numbered lines, each comprising one or more BASIC commands. These commands tell the computer what you want it to do; the line number which precedes them tells the computer not to obey the commands immediately, but to store them in RAM, and to execute them in order when the command **RUN** is given. RUN is a BASIC *keyword* i.e. one of the words in the MSX BASIC language. It can be used as a command within a program, but is usually used in direct mode.

The keyboard skills learnt in the previous section are essential for entering programs into the computer. Here is a short BASIC program for you to enter:

```
10 CLS
20 PRINT "Hello there"
30 print "I am your MSX computer."
40 END
```

This short program uses just three different keywords. The first of them, **CLS**, performs the same function as the <CLS> key: it clears the screen. **PRINT** is used to print something on the screen. Here, the messages enclosed in double quotation marks are to be printed. The use of PRINT will be explained further in Section 5.2. The final keyword, **END**, tells the computer that it has reached the end of the program and should now return to command mode to await further instructions.

Each line must be typed on the screen, then entered into the computer's memory by pressing <RETURN>. The lines can be entered in any order; the computer automatically sorts them into numerical order for you. Only the latest version of any numbered line is stored, so if you make a mistake in entering a line, either retype it or correct it, then press <RETURN> again to replace the incorrect version in memory with the correct version.

When you press <RETURN>, if there is a valid program line number at the start of the line (a number from 0 to 65529), all the characters on the line – regardless of the position of the cursor – will be entered as a program line. You must, therefore, delete any unwanted characters to the right of the cursor, using the delete key or <CTRL> and E, before pressing <RETURN>.

If you enter a program line by mistake, you can delete the whole line from memory by typing just the line number then <RETURN>, or the keyword **DELETE** then the line number then <RETURN>. The syntax for the keyword DELETE is:

DELETE <line no.>

to delete just a single line, or

DELETE <line no.>-<line no.>

to delete the two specified lines and any lines between them, or

DELETE -<line no.>

to delete all lines up to and including the specified line. Thus DELETE 50 will delete line 50; DELETE 10–100 will delete lines 10 and 100 and any lines with line numbers between 10 and 100; DELETE –30 will delete line 30 and any lines with line numbers lower than 30. The line numbers are *parameters* of the keyword DELETE. The broken brackets < > are used to indicate a parameter; they contain a description of the parameter required. In use, you must substitute the actual parameter for the broken brackets and their contents. Many keyword syntax examples will contain these broken brackets. Some will also contain square brackets, used to indicate optional parameters; the last two examples for DELETE could have been combined into:

DELETE [<line no.>]-<line no.>

When a syntax example contains round brackets, however, it will be because these are needed in the BASIC statement.

To delete a whole program, use the keyword **NEW**. This is one of the few keywords which does not require any parameters. Alternatively, you can reset or switch off the computer.

The **LIST** command enables you to check the lines you have entered. The syntax is:

LIST [<line no.>] [-<line no.>]

i.e. you can use LIST on its own to list the whole program, or specify a single line, a start line and/or an end line. LIST. will list the last line referred to by the computer. Function key <F4> will print LIST for you; <F8> will print LIST. (with an automatic RETURN).

If the listing cannot all be fitted on the screen at once, it will scroll upwards; press **<STOP>** to stop the listing at any point. This is a toggle key, so <STOP> again will restart the listing. You can use the <STOP> key to stop a program as well as a listing. There is also a **STOP** command which can be used within a program; to restart a program which has been stopped by a STOP command, use the command **CONT**. <CTRL> and <STOP> will stop the execution of any program or command and put the computer into command mode; CONT will restart execution after this too. <CTRL> and <STOP> can get you out of many sticky situations, so remember this combination!

To list a program on a printer instead of the screen, use **LLIST**, with the same syntax as for LIST. Do not use this command unless you have a printer connected to your computer!

If you entered the lines of our short program just as they were given at the start of this section, when you list the program you will notice that the computer has made one change. The keyword 'print' in line 30 was entered in lower-case letters, but it is listed in capital letters, though the words 'I am your MSX computer.' have not been altered. Keywords can be entered in capital letters, lower-case letters or a mixture of the two, but the computer always lists them in capitals. This is because they are stored in memory in token form, not as a series of characters.

When your listing shows that you have entered the program correctly, press <F5> to RUN it (or type RUN then press <RETURN>). The screen should display:

Hello there
I am your MSX computer.

```
OK
```

The 'OK' at the end is a message from the computer to tell you that it has finished and is ready for further instructions. The cursor should reappear directly below this. The program is still in memory, so you can RUN it again if you wish. If you did not obtain the correct screen display, LIST the program, edit it, then RUN again.

There is one more BASIC command to assist you with program entry. This is the **AUTO** command, which automatically generates line numbers. It can be typed in or obtained by pressing <F2>. The syntax is:

```
AUTO [<no.>], [<no.>]
```

The two numbers are both optional. The first specifies the first line number to be generated, the second specifies the interval between line numbers. If you want to specify the second only, remember to precede it with a comma to make your intentions clear. The default value of both is 10, so AUTO on its own will produce line numbers 10, 20, 30 etc. AUTO 50,5 will produce line numbers 50, 55, 60, 65 etc. A new number will be generated each time you press <RETURN>. If the number generated is that of an existing program line, it will be followed by an asterisk; just press <RETURN> to preserve the existing line, or delete the asterisk if you want to replace it with a new line. To exit from AUTO mode, press <CTRL> and <STOP> or <CTRL> and C.

Now you should be able to enter longer program listings from books and magazines, but before you spend hours typing, read the next section on saving programs so that you can preserve your work when you switch the computer off.

3.3 SAVING PROGRAMS ON CASSETTE

Cassettes provide a cheap and fairly simple means of saving your programs, both those you write yourself and those you type into your computer from printed listings. All the equipment you need is a standard cassette recorder and a lead to connect it to the computer. See Section 2.3 for instructions on connecting the cassette player to the computer.

If you have a REM socket on your cassette recorder and have plugged in the appropriate jack plug, the computer will be able to

control the cassette player's motor. If you put a cassette in the recorder and switch it to 'PLAY', you should find that nothing happens, because the computer has not turned the motor on. Type **MOTOR ON** <RETURN>, and the cassette should start to play. Type **MOTOR OFF** <RETURN>, and it should stop again. Instead of MOTOR ON and MOTOR OFF you can type just **MOTOR**, which will switch the motor on if it is off, and off if it is on. Alternatively you can remove the plug in the REM socket when you want to operate the recorder manually (to rewind a cassette, for example).

Ordinary audio cassettes can be used for computer programs. It is possible to buy special computer cassettes, but these are normally just unusually short audio cassettes. Using short cassettes makes it easier to find your programs again, particularly if you have saved more than one program on a single cassette.

Keep a careful record of the programs you save on each cassette and, if your cassette player has a tape counter on it, of the position at which each program starts. It is best to leave a gap between the programs, so that you can find the start of each one easily.

Cassettes are not indestructible, so if you are wise you will record each program on at least two cassettes. You could use a separate short cassette for each of your programs, and record all your back-up copies on a single long cassette. Commercial software is often supplied with a back-up copy on the reverse side of the cassette, but this is not particularly useful; if the tape is damaged in any way, it is likely that both sides will be unplayable.

When you are ready to save a program, select a cassette and wind it to the right position. If you are going to record the program on a new cassette, make sure that you have wound past the leader (the transparent, sometimes red, section at the start of the tape which cannot be used for recording). Set the volume control on the cassette recorder to maximum, and the tone controls (if any) to about half-way. Choose a name for the program. The name can be up to six characters long, and should be one which will remind you of what the program does, if possible. Make a careful note of the name, as you will need it again later.

There are three different commands which can be used to save programs on cassette. The one which is normally used for BASIC programs is **CSAVE**. The MSX can transmit data to the cassette tape at two different rates, 1200 or 2400 baud. The normal rate is 1200 baud, but for very long programs it is better to use the faster rate. The baud rate can be set by adding a parameter to the end of the CSAVE command: 1 for 1200 baud. If the parameter is omitted, a rate of 1200 baud will be assumed. The syntax for CSAVE is thus:

CSAVE"<program name>"[,<baud rate>]

The program name must always be enclosed in double quotation marks.

Set the cassette player to RECORD, then press <RETURN>. The cassette motor should start, and the program will be recorded. When it is finished, the computer will display an 'Ok' message, and if the REMOTE socket is being used, the motor will stop. (If not, stop the cassette recorder manually.)

Now you must check that the program has been recorded successfully. There is a special BASIC command which will compare the program recorded on tape with the one in the computer's memory, and let you know if they are the same. This command to verify a recording is **CLOAD?**. Rewind the tape to the start of the recording (using the MOTOR command if necessary), then type:

CLOAD?"<program name>" <RETURN>

and set the cassette recorder to PLAY. After a short while you should see

Found: (program name)

on the screen, followed – if the recording was successful – by 'Ok'.

If the program name you use in the verify command is not the same as the name under which the program was recorded, the computer will display the name of the program it finds recorded, then carry on searching through the tape to try to find the program whose name was used in the verify command. If you have entered the name wrongly in the verify command you must, then, press <CTRL> and <STOP> to stop this search and start again, making sure that you use the correct name this time. This incidentally provides you with a good way of finding out what programs are recorded on a cassette: just ask the computer to verify the recording of a non-existent program, and it will display the names of all the programs on the tape as it searches through them.

If your program was not recorded successfully, any one of three things can happen when you try to verify it: the computer can search through the tape without finding the program (as if you had entered the wrong name); you can get a 'Device I/O error' message; or you can get a 'Verify error' message. If this happens, stop the search if necessary by pressing <CTRL> and <STOP>, check the controls on the cassette

player, make sure that the EAR and MIC jack plugs are in the right sockets, and repeat the saving and verifying procedures. You can often get some idea of where the problems lie by listening to the tape. Remove the EAR plug to do this. If the program has been CSAVEd successfully, you should hear two bursts of a loud, high-pitched whistling sound, followed by a lower-pitched buzzing.

The CSAVE command saves the program in a tokenized format. It can be saved in a different format (ASCII code: see Section 5.3) by using the command **SAVE**. The format for this is:

SAVE"CAS:<program name>"

CAS: is a device descriptor, which tells the computer that the program is to be saved on cassette. SAVE is slower than CSAVE, but has two advantages: programs saved using SAVE can be merged (see Section 3.5), and they can also be RUN automatically when they are loaded (see Section 3.4).

The third save command, **BSAVE**, is used not for BASIC programs but for machine-code programs or sections of memory. The start and end addresses of the memory you want to save must be specified. For a machine-code program, the execution address can also be specified. The format for BSAVE is:

BSAVE "CAS:<program name>",<start address>,<end address>[,<execution address>]

Published listings of machine-code programs normally include detailed instructions on how to enter and save them. If you find a machine-code listing without instructions, you would be well advised to leave it alone until you have learnt some machine code yourself.

3.4 LOADING PROGRAMS FROM CASSETTE

When loading a program from cassette, you must use the load command which corresponds to the save command used to record it. The load commands for the three save commands are:

CSAVE – **CLOAD**
SAVE – **LOAD**
BSAVE –**BLOAD**

When loading commercial software, look carefully at the instructions to see which load command is to be used. With your own programs, you should make a note of the commands used when recording them, together with the program names.

To load a program: ensure that the cassette lead is properly connected, wind the tape to the beginning, type the appropriate load command then <RETURN>, then set the cassette recorder to PLAY. The computer should respond with 'Found: (program name)', then 'Ok'. If there is more than one program on the tape and the program you are loading is not the first, then you will get more than one 'Found' message.

The formats used for CLOAD and LOAD are:

CLOAD"<program name>"

or to load the first program on the tape, just CLOAD; and

LOAD"CAS:<program name>"

or to load the first program, LOAD"CAS:".

Programs saved using SAVE can be run automatically when they have been loaded if you add an R to the end of the load command:

LOAD"CAS:<program name>",R

The formats for BLOAD are as follows:

BLOAD"CAS:"

loads the first program on the tape;

BLOAD"CAS:<program name>"

loads the named program;

BLOAD"CAS:<program name>",R

loads then automatically RUNs the program;

BLOAD"CAS:<prog.name>",<no.>
or BLOAD"CAS:<prog.name>",R,<no.>

offsets the program from the location from which it was recorded by the amount specified.

If an execution address was specified when the program was BSAVEd, then the auto-run option will start execution of the program from this address.

If you accidentally use the wrong load command, don't worry; just press <CTRL> and <STOP>, rewind the tape, and try again.

Exercise 3.3

Select a short BASIC program, from this book or from another source. Enter it, and save it on cassette using CSAVE. Save it again using SAVE. Load both versions, and compare the times taken to load them.

3.5 MERGING PROGRAMS

When a new program is loaded, any program already in the computer's memory is normally obliterated. There are occasions, though, when it would be useful to load a new program without losing the old one. You may, for example, want to incorporate a program on cassette into a new program you are writing as a subroutine. This can be achieved using the **MERGE** command.

It is important to ensure, when using this command, that the programs which are being combined have different line numbers. Any lines in the program in memory which have the same line numbers as lines in the program being merged will be overwritten. The command **RENUM** can be used to change the line numbers of the program in memory, to prevent this from happening.

The format of this command is:

RENUM [<n>],[<m>],[<i>]

The number *n* is the new start line number, *m* is the old start line number and *i* is the increment between line numbers. The lines in the program, starting with line number *m*, are renumbered so that line *m* becomes line *n*, the next line becomes line $n + i$, the next line $n + 2i$ and so on. Any or all of the numbers *n*, *m* and *i* can be omitted; the default values are 10, the first line number in the program, and 10, respectively.

Example

```
 5 REM RENUM DEMO
10 REM This was line no. 10
```

```
15 REM This was line no. 15
20 GOTO 30
25 END
30 GOTO 25
```

RENUM changes this program to:

```
10 REM RENUM DEMO
20 REM This was line no. 10
30 REM This was line no. 15
40 GOTO 60
50 END
60 GOTO 50
```

Note that the line numbers in the GOTO statements have been changed; RENUM alters the numbers in all branching instructions, to preserve the order of execution of the program lines. The lines starting with REM, however, are comments, not BASIC commands, and so the numbers in these are unchanged.

RENUM 30, 15, 20 would change the original program to:

```
 5 REM RENUM DEMO
10 REM This was line no. 10
30 REM This was line no. 15
50 GOTO 90
70 END
90 GOTO 70
```

RENUM cannot be used to change the order of program lines; you cannot, for instance, renumber a block of lines at the end of a program to move them to the start. However, you can use it to create spaces into which new lines can be inserted.

Programs which are to be merged must be saved on cassette using SAVE, not CSAVE. Make a note of the line numbers when saving them, so that you will know how to renumber the programs with which they are to be merged. Load or type in the new program and renumber it if necessary. Then, to incorporate the program on cassette, type:

MERGE"CAS:[<program name>]"

If the program name is omitted, the first program on the tape which was saved in the correct format will be merged.

The merging procedure is the same as the loading procedure described in Section 3.4.

4

Writing a BASIC Program

4.1 WHAT IS A PROGRAM?

A *program* is a series of instructions, written in the computer's own BASIC language, which can be stored in the computer's memory and then executed in the prescribed order whenever the instruction RUN is given. Using the computer in *direct mode*, i.e. entering commands to be executed immediately, is convenient if you want to perform just a simple operation involving only one or two BASIC commands, but for more complex tasks a program is essential.

Whenever the <RETURN> key is pressed, the computer checks what has been typed in and takes one of three possible courses of action. If the input starts with a positive number between 1 and 65529, it is treated as a program line and stored in memory. If it is a valid BASIC command (or a series of commands separated by colons) then it is executed immediately. Otherwise the computer responds with 'Syntax error'.

A single program line can contain more than one instruction. A program will take up less space in the computer's memory if several instructions are put in each program line, instead of a separate line being used for each instruction. The instructions in a *multi-statement line* must be separated by colons, but no colon is necessary after the last instruction in the line. The length of a line is limited to 255 characters.

Example

```
10 CLS
20 PRINT "Think of a number."
30 PRINT "Multiply it by 10, then add 50."
40 PRINT "Divide by 5, subtract 2."
50 PRINT "Now divide by 2."
60 PRINT "Take away the number you first thought of."
```

```
70 PRINT "The answer is 4!"
80 END
```

This program could be rewritten as:

```
10 CLS:PRINT"Think of a number.":PRINT"Multiply it by 10,
   then add 50.":PRINT"Divide by 5, subtract 2."
20 PRINT"Now divide by 2.":PRINT"Take away the number you
   first thought of.":PRINT"The answer is 4!":END
```

The original eight program lines have been condensed into just two lines.

The program lines are stored in memory in numerical order. When the instruction to RUN the program is given, the computer starts with the lowest numbered line, unless RUN is followed by a line number, in which case it starts from the specified line. The lines are executed in numerical order, unless the program contains instructions which change the order of execution. If the program contains an instruction which does not make sense, or which cannot be executed for some reason, the computer will stop when it reaches that command and print out an error message, specifying the program line in which the problem arose. Otherwise execution will continue until the computer reaches the command END, or until there are no more commands to be executed.

There are several different BASIC commands which change the order of execution of a program. Some of these are unconditional commands, instructing the computer to transfer execution to a specified program line under all circumstances. Others are conditional: the computer can be instructed to test the value of a variable or expression, and to branch to a different part of the program, depending on the value obtained. Conditional branches and loops are discussed in Chapter 9; unconditional branches are discussed later in this chapter.

4.2 FLOWCHARTING

Before you start writing a computer program, you must first decide what you want the program to do. We will take as an example a simple game program, noughts and crosses. The game must be broken down into small stages. These need not correspond to BASIC commands – each one may involve a considerable amount of programming. The stages in this game are:

1. Explain the rules.
2. Request players' names.
3. Display playing grid on screen.
4. Ask first player to enter a cross on grid.
5. Check to see if there is a line of crosses.
6. If there is a line of crosses, winner=first player: go to stage 12.
7. Check to see if grid is full. If so, winner=no one: go to stage 12.
8. Ask second player to enter a nought on grid.
9. Check to see if there is a line of noughts.
10. If there is a line of noughts, winner=second player: go to stage 12.
11. Go back to stage 4.
12. Display result.
13. End.

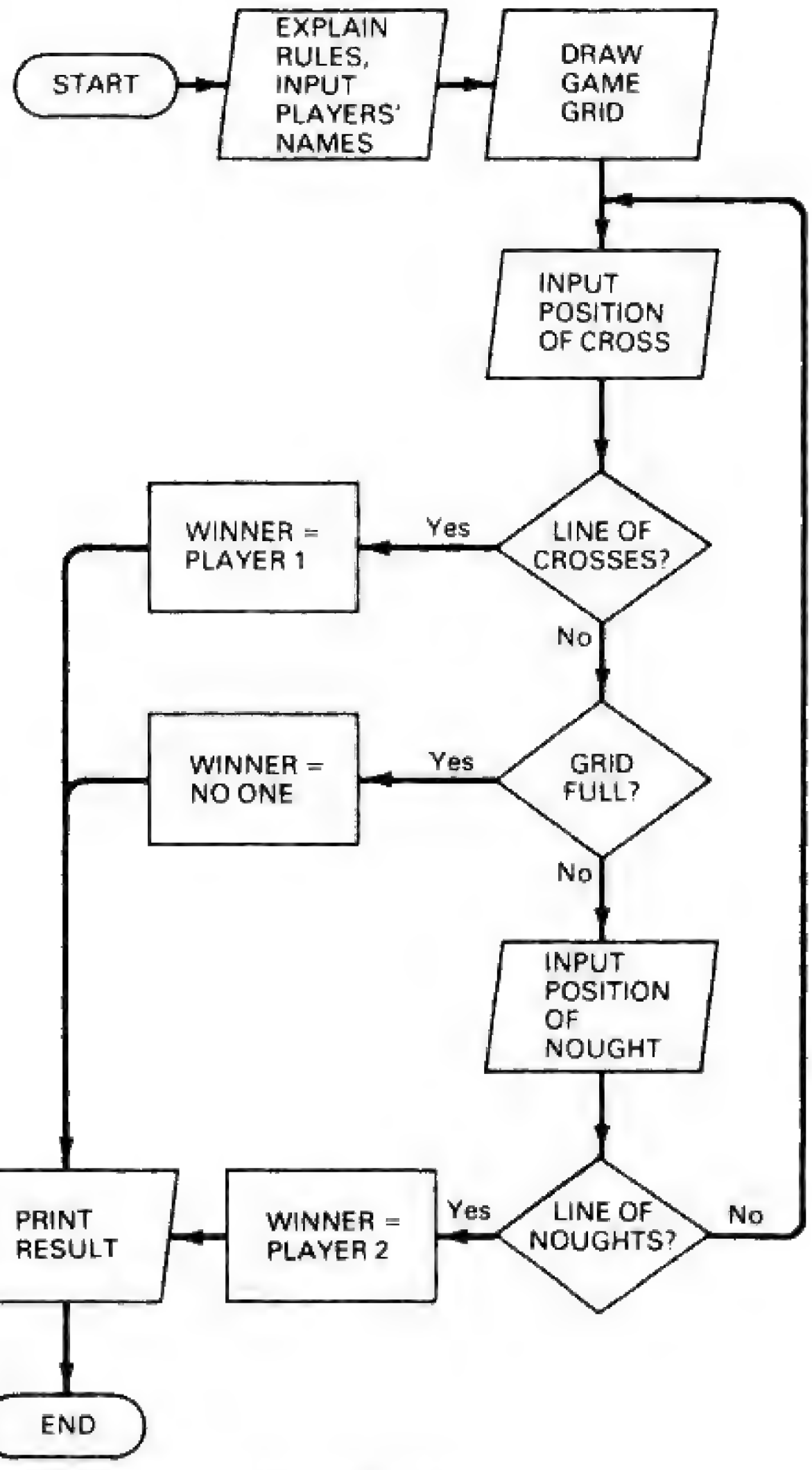


Figure 3 *Noughts and crosses flowchart*

The structure of the game can be made clearer by displaying the stages in the form of a flowchart. Each stage is put inside a little box. Different shapes of boxes are used for different types of stages: a rectangular box is used for processing operations, a parallelogram for input/output, a diamond shape for decision points where there are two or more possible next stages, and so on. These boxes are joined by arrows, to show the order in which they are to be executed. The arrows leading from decision boxes are labelled, to show the conditions under which they will be followed. Figure 3 shows the flowchart for the noughts and crosses game.

Once the flowchart has been drawn up, the contents of each box can be translated into BASIC commands. The task of writing a program to play noughts and crosses has been transformed into a succession of smaller and far less daunting tasks, and the resultant program can be expected to have a clear, well-defined structure, with a separate *routine* (series of instructions) for each of the stages in the game.

The planning and writing of several programs is described in detail in Chapter 19.

4.3 UNCONDITIONAL JUMPS

The BASIC command **GOTO** produces an unconditional jump to a specified line. The syntax is:

GOTO <line no.>

If a BASIC program has been well thought out and is properly structured, there should be very little need to use unconditional jumps. The routines of which the program is composed should follow one another in a logical manner, each one leading naturally to the next. The use of excessive numbers of unconditional jumps makes programs hard to follow, and greatly complicates the process of *debugging* (finding and eliminating any errors). You should therefore use GOTOs very sparingly.

One legitimate use of GOTO is to make the computer execute part or all of a program over and over again: just put a GOTO at the end of the section to be repeated, transferring execution back to the beginning.

Example

```
10 PRINT"This program will give you the sum and difference of
   two numbers."
20 GOTO 50
30 PRINT"Sum=";A+B
40 GOTO 80
50 INPUT"First number";A
60 INPUT"Second number";B
70 GOTO 30
80 PRINT"Difference=";A-B
90 GOTO 50
```

This program contains too many GOTOs; the order of the program lines is illogical. It should have been written like this:

```
10 PRINT"This program will give you the sum and difference of
   two numbers."
20 INPUT"First number";A
30 INPUT"Second number";B
40 PRINT"Sum=";A+B
50 PRINT"Difference=";A-B
60 GOTO 20
```

Now it is obvious what the program does, and the function of the one remaining GOTO is clear. (Note that this version is also much shorter than the first.)

GOTO can also be used to make the computer 'stick' at a particular point:

```
100 GOTO 100
```

You will see a statement like this at the end of many of the graphics programs in later chapters. If the computer reaches the end of a graphics program, the graphics display will disappear and be replaced with an 'Ok' message; the graphics can only be held on the screen by putting in a command which prevents the end of the program from being reached. When the graphics are no longer required, pressing <CTRL> and <STOP> will put the computer back into command mode.

4.4 SUBROUTINES

If the same sequence of instructions is used more than once in a program, then these instructions can be put in a *subroutine*. Execution is transferred from the main program to the subroutine by a **GOSUB** command. The syntax for this is:

GOSUB <line no.>

where the line specified is the first line of the subroutine.

The subroutine must end with a **RETURN** command. When the computer encounters a GOSUB command, it puts a number which indicates the point it had reached in the program – the *address* of the GOSUB command – into a special section of memory called the *stack*. When the RETURN command is reached, this address is retrieved from the stack and used to return execution to the instruction following the GOSUB command. The stack can hold more than one number; the order in which numbers are retrieved from it is the reverse of the order in which they were added to it, i.e. the last number to be added to the stack is the first to be taken from it. This means that subroutines can be *nested* inside one another: execution can be transferred from the main program to a first subroutine, then from the first subroutine to a second subroutine, and so on. When a RETURN command is encountered, execution will be transferred to the instruction following the last GOSUB command to be executed.

Example

```
        .
        .
100  GOSUB 500
110  .
200  GOSUB 600
210  .
        .
400  END
500  (First subroutine starts here)
        .
550  GOSUB 600
560  .
        .
590  RETURN
600  (Second subroutine starts here)
```

```
690 RETURN
```

This outline program contains two subroutines. Both are called from within the main program routine, and the second subroutine is also nested within the first. When execution reaches line 100, the address of this GOSUB command will be put on the stack and execution will be transferred to line 500. When line 500 is reached, the address of the GOSUB command in this line will be added to the stack and execution will be transferred to line 600. When line 690 – RETURN – is reached, the last address to be added to the stack will be retrieved. This is the address of the GOSUB command in line 550, so execution will be returned to the following command: line 560. At line 590, the remaining number on the stack will be retrieved, transferring execution to line 10. Next the address of the GOSUB command in line 200 will be added to the stack and execution will be transferred to line 600; finally, at line 690, retrieving this number will return execution to line 210.

In the noughts and crosses game (Section 4.2), some processes were repeated for each of the two players: entering a piece on the grid, and checking for a complete line. These sections of the program could be written as subroutines. The values of variables can be passed from the main program to subroutines, so the fact that noughts were being entered and checked for one player, and crosses for the other, would not prevent this; a variable could be used to represent the player's symbol, and this variable could be assigned the value "X" before the subroutine was called for the first time, and "O" before it was called for the second time. Here is the relevant part of the program:

```
        .
        .
100  LET S$="X"
110  GOSUB 500
120  GOSUB 600
130  .
     (This section checks to see if grid is full)
     .
200  LET S$="O"
210  GOSUB 500
220  GOSUB 600
230  .
     (Remainder of program)
```

```
      .
490 END
500 (Subroutine to enter piece on grid)
590 RETURN
600 (Subroutine to check for complete line)
      .
690 RETURN
```

It is usually convenient to group all the subroutines used in a program together at the end. You must make sure that the main program ends with an END command, or with a GOTO command; otherwise when the computer has finished executing the program, it will go on to execute the first subroutine, and will stop with a 'Return without Gosub' error when it reaches the RETURN and does not have anywhere to return to.

4.5 COMMENTS

Whenever you write a computer program you should make a note of the name of the program and what it does, draw up a flowchart (unless the program is very simple) and make a list of the variables used. This documentation will come in very useful if you later want to edit or amend the program, or if you want to use some of the routines you have written for it in a different program. It would also be very useful if you could label the program listing itself, putting in comments at various points, marking the start and end of each routine so that it could be identified easily and so on. These labels and comments could be saved on tape with the rest of the program; they would provide a useful back-up to the written documentation and, unlike the written documentation, could not be mislaid. **REM** (remark) statements provide you with a way of putting such comments and labels into your programs.

When the computer encounters the word REM in a program line, it ignores the remainder of the line and goes straight on to the next line. The comments which follow REM are not checked for syntax, and so need not make sense in BASIC. They will be ignored even if they are valid BASIC commands, so if you put a REM in a multi-statement line, make sure it is the last statement in the line!

You should try to avoid using the line numbers of REM statements in GOTO, GOSUB or other branching commands. If you were to find yourself running short of memory space, you might decide to delete

some or all of the comments in your program; if this meant that the program contained references to non-existent lines, execution would be stopped by 'Undefined line number' errors.

Most programs start with one or more REMs, containing information such as the program name, the programmer's name and the date when the program was written. Lines of asterisks (in REM statements, of course) may be used to make routines stand out.

Example

```
 10 REM Outline program to show the use of REMs
 20 REM By Margaret Norman
 30 REM May 1985
 40 REM ********************************
 50 REM********Title screen ***************
 60 REM ********************************
 70 .
    .
100 REM ********************************
110 REM********Main section of program ***********
120 REM ********************************
130 .
    .
200 GOSUB 500
    .
    .
300 GOSUB 700
    .
    .
400 END
470 REM ********************************
480 REM ********First subroutine*****************
490 REM ********************************
500 .
    .
600 RETURN
670 REM ********************************
680 REM ********Second subroutine  ***************
690 REM ********************************
700 .
    .
800 RETURN
```

An apostrophe (') can be used as a substitute for the word REM. If you use these when entering a program, to speed up the entry process, the computer will replace them with REMs when the program is listed.

Exercise 4.1

List the line numbers of this program, in the order in which they will be executed:

```
 10 REM Program tracing exercise .
 20 GOTO 80
 30 GOSUB 60
 40 PRINT"Do not structure your programs like this!"
 50 RETURN
 60 PRINT"This is a very messy program.":REM Subroutine 2
 70 RETURN
 80 REM by Margaret Norman: GOTO 110
 90 GOSUB 30
100 GOTO 130
110 GOSUB 60
120 GOTO 110
130 END
```

5 Printing

5.1 THE TEXT MODES

When the computer is first switched on, a text screen is displayed which is 40 characters wide and 24 characters long, i.e. there is room for 24 rows of 40 characters each; this is text mode 0. There is an alternative text screen which is 32 characters wide and 24 characters long; this is text mode 1. The **SCREEN** command is used to switch from one to the other. This command has other functions which will be explained later; to change text modes, however, all you need to know is that SCREEN 0 will select text mode 0, and SCREEN 1 will select text mode 1.

The rows and columns of both text screens are numbered, starting from 0 at the top left-hand corner. Thus the left-hand column is column 0, and the right-hand column is column 39 for mode 0 and column 31 for mode 1. The top row is row 0, and the bottom row (on which the function key list is normally displayed) is row 23.

The command **WIDTH**(no.) can be used to set the number of screen columns which are to be used. The number can range from 1 to 40 for mode 0, and from 1 to 32 for mode 1. The most central columns are selected by the width command, i.e. if you select SCREEN0:WIDTH36 the two leftmost columns and the two rightmost columns of the 40 column mode 0 screen will not be used. As some televisions will not display all the text columns, select a screen width which fits comfortably on your television screen. You will notice that when the width of the screen is changed, the screen is automatically cleared. The default width values are 37 for mode 0, and 29 for mode 1.

The character positions in mode 1 are wider than those in mode 0, as there are fewer of them. When the graphics characters are displayed in mode 0, a strip off the right-hand edge is missing from some of them; in mode 1, the whole characters can be displayed. The letters and numbers

are all designed so that they will fit in the mode 0 character positions.

5.2 PRINTING

The command **PRINT** can be used to print characters on either of the text screens. This command is followed by the actual characters to be printed, enclosed in double quotation marks, or by a constant, variable or expression whose value is to be printed. You can substitute a question mark for the word PRINT if you wish.

Examples

```
PRINT "Have a nice day." <RETURN>
Have a nice day.
Ok

A=5.87:PRINT A <RETURN>
 5.87
Ok

B$="Abracadabra":?B$ <RETURN>
Abracadabra
Ok
```

The PRINT command can be used in direct mode to obtain the results of calculations, i.e. it enables you to use the computer like a calculator:

```
?54.6*(8+327.21)/45
 406.72146666667
Ok
```

The operators and functions available are described in Chapter 7.

When PRINT is used within programs, it is usually used in conjunction with commands to format the output, i.e. to arrange it nicely on the screen. The computer starts printing at the cursor position. If you want to place the output at a particular position on the screen, you can use the **LOCATE** command to move the cursor before printing. This takes the form:

```
LOCATE [<X co-ordinate>],<Y co-ordinate>
```

where the X co-ordinate is the number of the column to which the cursor is to be moved, and the Y co-ordinate is the row number. The X co-ordinate (but not the comma following it) may be omitted, in which case only the row is changed.

Type in and RUN this program:

```
10 SCREEN0:WIDTH30
20 LOCATE 10,12
30 PRINT "In the middle"
40 GOTO 40
```

Line 10: selects text mode 0, screen width 30 columns.
Line 20: moves the graphics cursor to column 10, row 12.
Line 30: prints message at the cursor position.
Line 40: the computer will 'stick' here.

The words 'In the middle' should appear in the middle of the screen. This is not followed by the usual 'Ok' message, because the computer cannot get to the end of the program. To stop the program running, press <CTRL> and <STOP>.

The appearance of the screen display produced by this program was rather spoiled by the list of key words at the bottom of the screen. The command **KEY OFF** will remove these. When you want them back again, the command **KEY ON** will restore them. You cannot print anything on the bottom line of the screen unless you first execute KEY OFF. Type in another program line:

```
15 KEY OFF
```

The computer will automatically insert this line into its proper position in the program. Now when you RUN the program, the screen should display nothing but the words 'In the middle'.

If you want to print more than one item on a single line, you can prevent the cursor from being moved to the start of the next line when the printing has been completed by putting a semicolon at the end of the PRINT statement. A single PRINT command can be used to print more than one item; if you separate the items by semicolons, they will all be printed on the same line. If you separate them by commas instead, they will be printed in two columns.

Example

```
10 A=34.6
20 B=2.87
```

```
30 SCREEN1:WIDTH28
40 ?"I can";
50 ?"do sums!"
60 ?"A=",A,"B=",B
70 ?"A+B=",A+B
80 END
```

Line 10: assigns a value to the variable A.
Line 20: assigns a value to the variable B.
Line 30: selects text mode 1, screen width 28 columns.
Line 40: prints 'I can', leaving the text cursor on the same line.
Line 50: prints 'do sums!' immediately after 'I can', moves text cursor to start of next line.
Line 60: prints 'A=', moves to the middle of the row and prints the value of A, moves to the start of the next row and prints 'B=', moves to the middle of this row and prints the value of B, then moves cursor to start of next line.
Line 70: prints 'A+B=' followed by the value of A+B, then moves cursor to start of next line.
Line 80: end of program (computer prints 'Ok' message).

If you RUN this program, you should get:

```
I can do sums!
A=          34.6
B=          2.87
A+B=        37.47
Ok
```

Commas can be used to arrange the output in two columns; if you want more columns than this, the **TAB** command, which can only be used in conjunction with PRINT, can provide them. The <TAB> key moves the text cursor to the next preset tab position; the positions are at 8 column intervals across the screen. The TAB command offers more flexibility than this, as it can be used to specify the actual column to which the cursor is to be moved. The syntax is:

PRINT TAB (<column no.>**)**

Remember that the leftmost column is column 0, so if you start a PRINT statement with TAB(*n*) there will be *n* blank spaces before the first item to be printed. Several TABs can be used in a single PRINT statement, to arrange the output in two or more columns.

The command **SPC**, which can also only be used in a PRINT statement, inserts a specified number of spaces. The syntax is:

PRINT SPC(<no. of spaces>)

The number of spaces must be between 0 and 255. Spaces to be printed can also be put into character strings, of course, by enclosing them in double inverted commas, but when more than two or three spaces are required the SPC command makes the number of spaces clearer, as well as taking up less room in the program.

Example

```
10 PRINT TAB(3)"Name"TAB(17)"Score"
20 PRINT TAB(3) "Name"SPC(10)"Score"
30 PRINT "      Name                    Score"
```

Line 10: moves the cursor to column 3, prints 'Name', moves the cursor to column 17, prints 'Score', then moves the cursor to the start of the next line.

Line 20: moves the cursor to column 3, prints 'Name' then ten spaces then 'Score', then moves the cursor to the start of the next line.

Line 30: prints ' Name Score', then moves the cursor to the start of the next line.

This program illustrates three different ways of producing the same printed output.

PRINT USING enables you to format the numbers or character string to be printed. The following options are available for formatting numbers:

1. Specify the position of the decimal point, and the number of digits before and after it. A + sign is ignored; a – sign counts as one digit. A hash sign is used to represent each digit:

```
PRINT USING "###.##"; 2.753
 2.75
```

The same format may be used for more than one number:

```
PRINT USING "###.##";73,54.9,–2.8
 73.00 54.90 –2.80
```

Note that here the comma separating the numbers does not cause them to be printed in two columns as with a normal PRINT command.

Numbers are rounded if necessary. If they will not fit in the format specified, a per cent sign is printed in front of them:

```
PRINT USING "##.#"; 100.58
%100.6
```

2. Print a + **or** – sign before or after the number. A plus sign is included in the format string at the appropriate position:

```
PRINT USING "+##.##";5.78,–12.4
 +5.78–12.40
PRINT USING "##+";52.9
53+
```

3. Print the number in exponential format. Four exponential signs are added, to allow space for E+xx:

```
PRINT USING "##.##^^^^";187.3
 1.87E+02
```

4. Print a dollar sign or pound sign before the number:

```
PRINT USING "$##.##";–5.3
  $–5.30
```

5. Insert a comma between each three digits to the left of the decimal point. A comma is included somewhere before the decimal point:

```
PRINT USING "#,##########.##";3456789.78
3,456,789.78
```

6. Fill any leading spaces with asterisks. Two asterisks are included, each representing one digit:

```
PRINT USING "**#.##";3.4,52
**3.40*52.00
```

Various combinations of these options are also permissible.

The following options are available for formatting strings of characters:

1. Print the first character only. An exclamation mark is used:
   PRINT USING "!"; "Read", "Only", "Memory"
   ROM

2. Output a specified number of characters, truncating strings or adding spaces as necessary. The first and last characters are represented by slashes, those in between by spaces:

   PRINT USING "\ \";"ABC","UVWXYZ"
   ABC UVWXY

3. Insert the string into another specified string. Its position is given by &:

   PRINT USING "TODAY IS &DAY";FRI
   TODAY IS FRIDAY

Just as programs can be listed on a printer by prefixing the command list with an L, so printed output can be directed to a printer instead of the screen by prefixing PRINT AND PRINT USING with an L: **LPRINT**, **LPRINT USING**. TAB and SPC can be used with these commands in the same way as with normal PRINT commands. You can, of course, only use LPRINT and LPRINT USING if you have a printer connected to your computer!

5.3 THE ASCII CODES

The characters available on an MSX computer all have different code numbers, running from 0 to 255. The codes used are ASCII codes; this stands for American Standard Code for Information Interchange. The characters are shown together with their codes in Appendix A of this book.

The code number of a character can be obtained by using the function **ASC**. The character must be enclosed in double quotation marks inside brackets. If the quotation marks contain more than one character, only the code of the first character will be returned.

Example

```
PRINT ASC("A")
65
```

```
PRINT ASC("ANYTHING")
65
```

The function **CHR$** does the opposite to ASC: it produces a character, given its code number. It can be used to print graphics characters on the screen:

```
PRINT CHR$(65)
A
```

This works well with code numbers between 32 and 255, but with numbers from 0 to 31 some odd things happen. The codes from 0 to 31 are also used as *control codes*. When you ask the computer to print a control code character, instead of printing a character it will perform an operation. Most of the control codes move the cursor around the screen. Control code 13 is equivalent to <RETURN>; it gives you a way of using <RETURN> within a program, should you want it! Control code 7 produces a beep. The control codes are listed in Appendix E.

But there has to be some way of printing the graphics characters with codes from 0 to 31 – and this is it:

```
PRINT CHR$(1)+CHR$(n+64)
```

will print graphics character *n*, where *n* is between 0 and 31.

Using CHR$ functions in place of graphics characters will make your programs easier to follow. It is particularly important to use functions, not characters, if you intend listing your programs on a printer which cannot handle the graphics characters.

Example

```
10  SCREEN1:WIDTH30
20  LOCATE 10,5
30  PRINT CHR$(1)+CHR$(66)
40  PRINT SPC(9)CHR$(47);CHR$(219);CHR$(92)
50  PRINT SPC(10)CHR$(219)
60  PRINT SPC(10)CHR$(198);CHR$(198)
```

Line 10: selects text mode 1, screen width 30 columns.
Line 20: moves the cursor to column 10, row 5.
Line 30: prints graphics character 2, moves the cursor to the start of the next line.

Line 40: prints nine spaces then three graphics characters, then moves the cursor to the start of the next line.
Line 50: prints ten spaces then graphics character 219, then moves the cursor to the start of the next line.
Line 60: prints ten spaces then graphics character 198 twice, then moves the cursor to the start of the next line.

When you RUN this program, a rather odd little figure should be printed on the screen.

If you want to print a whole string of one particular character, you need not use repeated CHR$ functions; instead, you can use the function **STRING$**. This takes the form:

STRING$(<no. of characters>,<ASCII code>)

Example

```
10  SCREEN1:WIDTH 20
20  LOCATE 5,5
30  PRINT STRINGS$(10,CHR$(215))
40  PRINT TAB(5)CHR$(215)SPC(8)CHR$(215)
50  PRINT TAB(5) STRING$(10,CHR$(215))
```

Line 10: selects text mode 1, screen width 20 columns.
Line 20: moves the cursor to column 5, row 5.
Line 30: prints graphic character 215 ten times (then moves the cursor to the start of the next line).
Line 40: moves the cursor to column 5 then prints graphics character 215, eight spaces, graphics character 215.
Line 50: moves the cursor to column 5 then prints graphics character 215 ten times.

A small box of chequerboard-pattern squares is printed on the screen.

Exercise 5.1

Write a program which would print this letter heading on a printer:

Tel 01-234-5678
1 New Road
Any Town
Blankshire
England

NB Do not RUN the program unless you have a printer!

Exercise 5.2

Write a program to print this table on the screen, using TAB and PRINT USING to arrange the numbers in neat columns:

n	*n*^2	*n*^3	*n*^4
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625

6

Number Systems, Variables and Constants

6.1 NUMBER SYSTEMS

Most numerate human beings today usually use the *decimal number system*, i.e. a system with base 10 using the ten digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The reasons for this are more biological than logical: we happen to have ten fingers. Computers store numbers using switches which have two possible states, on or off, so they use the *binary* system, i.e. a system with base 2 using the two digits 0 and 1.

Binary digits, or *bits*, are grouped in sets of eight within the computer's memory; each memory location contains 8 bits, or 1 *byte*. This makes it convenient to use the *octal* and *hexadecimal* systems as well: 1 byte can be expressed as a four-digit octal number, or a two-digit hexadecimal number. Octal has a base of 8 and uses the eight digits 0, 1, 2, 3, 4, 5, 6 and 7. Hexadecimal, called hex for short, has a base of 16 and uses as its sixteen digits the numbers 0 to 9 and the letters A, B, C, D and E.

The numbers 0 to 20 (decimal) would be represented in these four systems as shown:

decimal	*binary*	*octal*	*hex*
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14

Binary, octal and hex numbers are distinguished by preceding binary numbers with **&B**, octal numbers with **&O** and hex numbers with **&H**. No prefix is used for decimal numbers. Thus the number 13 (decimal) could be written as 13, &B1101, &0115 or &HD.

The computer can only handle binary, octal and hex *integers* (whole numbers) within the range –32768 to 32767 (decimal). This is because 2 bytes (16 binary digits) of memory are used to store the numbers. The highest positive number which can be represented is &B0111111111111111: the leftmost bit is always zero for positive numbers. Negative numbers are represented in *two's complement* format, by the number which, when added to the corresponding positive number, will give a result of zero. The two's complement representation of –4 is &B1111111111111100; add &B100, and you get &B10000000000000000 (a 1 and sixteen 0s). The 1 disappears, as only sixteen digits are used. To determine the two's complement representation of a negative number: the leftmost bit is always 1; the other fifteen bits represent 32768 plus the number being represented.

Example

To determine the two's complement representation of –73 (decimal):

```
  32768+(-73)=32695
             =&B0111111110110111
        -73=&B1111111110110111
Check:       1111111110110111
           +          1001001      (+73)
           ------------------
            (1)0000000000000000
```

Don't worry if this seems rather complicated – the computer can convert numbers from one base to another for you!

To Convert Binary, Octal or Hex Numbers to Decimal

The computer always uses decimal numbers unless instructed otherwise, so if you assign a binary, octal or hex number to a variable and PRINT its value, or PRINT the number directly, the answer will be given in decimal.

Examples

Type in:

```
PRINT &HA8 <RETURN>
```

The computer should respond with:

```
 168
Ok
```

Or using a short program:

```
10 LET X=&B1101
20 PRINT X
30 END
```

RUN this, and you should get:

```
 13
Ok
```

To Convert Decimal Numbers to Binary, Octal or Hex

The special functions **BIN$**, **OCT$** and **HEX$** are provided to convert decimal numbers to binary, octal and hex respectively. The *argument* of the function (the number on which the function acts) must be a decimal integer, a variable or an expression whose value is within the range – 32768 to 32767.

Examples
Type in:

```
PRINT BIN$(5-23) <RETURN>
```

The computer should respond with:

```
1111111111101110
Ok
```

Or using a program:

```
10 LET A=2521
20 PRINT OCT$(A)
30 END
```

RUN this, and you should get:

```
4731
Ok
```

A decimal/hexadecimal conversion table is given in Appendix D.

Exercise 6.1

Convert the following decimal numbers to binary, octal and hex:

(a) 572
(b) 3144
(c) –2197

Exercise 6.2

Convert the following numbers to decimal:

(a) &B101100010
(b) &0374
(c) &HD2C

Exercise 6.3

Convert &H4E to binary.

6.2 VARIABLES

Data can be stored in the computer's memory under variable names. When you first introduce a variable, in a program or in direct mode, the computer assigns a small area to that variable and stores the value you have assigned to the variable in that area. Variables are so called because their value can be varied; when you assign a new value to the variable, the computer overwrites the old value with the new one.

The **LET** command is used to assign a value to a variable:

```
LET X=5
```

The LET can be omitted, leaving just

```
X=5
```

This looks like an equation, but it is not. What it means is: assign the value of the number or expression following the equals sign (here 5) to the variable preceding it (here X). The expression following the equals sign can include the variable itself:

```
A=A*2
```

means take the number stored in the memory location allocated to the variable A, multiply it by 2 and return the result to the same memory location.

Variable names must obey certain rules:

1. They must begin with a letter.
2. They may be of any length, but the computer will only recognize the first two characters; i.e. COUNTER, CO and CON will all be regarded as the same variable.
3. Capital and lower-case letters are not distinguished, so AB, aB, Ab and ab all represent the same variable.
4. The initial letter may be followed by letters or numbers, but not by other symbols.
5. The name must not include a reserved word or BASIC keyword (see Chapter 18 for a summary of keywords).

Every variable is of a specific *type*; the variable type determines the type of data which can be assigned to the variable. The type can be declared using a special BASIC statement, or can be implied by the presence or absence of a suffix after the variable name. Assigning the wrong type of data to a variable, or using the wrong type of variable with an operator or function, will produce a 'Type mismatch' error.

The variable types supported by MSX BASIC are:

1. *Double-precision variables* Any variable whose type is not declared is assumed to be a double-precision variable. Double-precision numbers are stored to fourteen decimal digits, and so can be used to perform very accurate calculations.
2. *Single-precision variables* When such great accuracy is not required, single-precision numbers can be used instead. These are stored to six decimal digits. They take up less memory space than double-precision numbers, and calculations can be performed significantly faster with them.
3. *Integer variables* Whole numbers in the range –32768 to 32767 (the same range as for binary, octal and hex numbers) can be assigned to integer variables. Real numbers assigned to integer variables are automatically rounded down to the nearest integer, so 3.25 becomes 3, and –56.4 becomes –57. Integers take up very little memory space, and calculations involving them can be performed very quickly.
4. *String variables* Any non-numeric data must be assigned to a string variable. These can contain up to 255 different characters, which may include numbers, letters, graphics characters, or any of

the other symbols in the MSX character set. The characters assigned to a string variable are normally enclosed in *double quotation marks* ("string").

The suffixes and declarations used to declare these variable types are:

variable type	*suffix*	*declaration*
double precision	#	**DEFDBL**
single precision	!	**DEFSNG**
integer	%	**DEFINT**
string	$	**DEFSTR**

Type declarations are followed by one or more letters; all variables whose names begin with one of these letters are declared to be of the stated type.

Suffixes take precedence over type declarations.

Example

```
 10 DEFINT A,B
 20 DEFSTR C
 30 A1=3.14
 40 DX=2.25
 50 Comment="I am a string variable"
 60 B$="Another string"
 70 PRINT A1
 80 PRINT DX
 90 PRINT Comment
100 PRINT B$
110 END
```

Here variables whose names start with A or B are declared to be integers, and variables whose names begin with C are declared to be strings. Thus A1 is an integer variable, and Comment is a string variable. As suffixes take precedence over declarations, B$ is a string variable. DX is not declared or suffixed and so, by default, is a double-precision variable.

The functions BIN$, OCT$ and HEX$ introduced in Section 6.1 are suffixed by the dollar sign used to identify string variables. This is because the computer automatically uses a decimal format for numeric data. If numbers are to be stored in a non-decimal format, then they must be assigned to string variables rather than numeric variables. The statement.

```
LET A=BIN$(253)
```

will produce a 'Type mismatch' error unless A has been declared a string variable.

String variables are all stored in a special area of the computer's memory, the size of which is initially set to only 200 bytes. If you think you will need more *string space* than this, you can use a **CLEAR** statement to increase the size of this area. For example,

```
CLEAR 400
```

allocates 400 bytes of memory to string storage. If you specify too little space, you will get an 'Out of string space' error. CLEAR does not enable you to assign more than 255 characters to an individual string; any attempt to assign more characters to a string will produce a 'String too long' error.

The CLEAR command also clears the variable storage area, so it erases the values of all variables and undimensions all arrays (see Section 6.3). It is therefore best used right at the start of a program.

A second parameter can be added to the CLEAR command to reserve memory space for machine-code programs: see Chapter 16.

Exercise 6.4

Which of the following are valid variable names?

(a) A$
(b) PRINTER
(c) 1A!
(d) F2%
(e) C*D

Exercise 6.5

```
10 DEFINT A
20 A1=4.76
30 A$="ABC"
40 AX!=32.459263174
```

Give the variable types of:

(a) A1
(b) A$
(c) AX!

6.3 ARRAYS

An *array* is a group of variables of the same type which have the same name, and are distinguished from one another by one or more *reference numbers* in brackets after the name. The number of different reference numbers is called the *dimension* of the array. You can think of a one-dimensional array as a collection of variables arranged in a column and numbers 0, 1, 2, 3 etc. (The numbering always starts with 0.) A two dimensional array is like a grid arrangement, with the reference numbers giving the row and column positions of the variable in the grid respectively. The maximum dimension is 255, but you are unlikely to need such a large array as that.

Arrays can be composed of any of the types of variable described in Section 6.2. String arrays are very useful for storing names and addresses: these can be put in a two-dimensional array, with one column for names, another for addresses and as many rows as necessary.

The individual variables in an array are known as its *elements*. If you want to use an array with more than eleven elements, you must tell the computer to reserve memory space for it by using a **DIM** statement. The command DIM is followed by the array name, then the maximum size of each dimension less one (because there is a 0th element) in brackets, separated by commas if there is more than one dimension. For example, to set up a two-dimensional string array with twenty rows and three columns:

```
DIM A$(19,2)
```

A single DIM statement can be used to set up several arrays; the array names must be separated by commas. To set up three different one-dimensional, fifteen-element numeric arrays:

```
DIM X(14),Y(14),Z(14)
```

If an array element subscript (reference number) is used which is greater than the number specified in the DIM statement, or greater than 10 if no DIM statement was used, then a 'Subscript out of range' error will be produced.

An array can be erased by the **ERASE** command. This is followed just by the array name(s):

```
ERASE A,B$
```

If you want to change the size of an array, you must erase the original array then redimension it. Any attempt to redimension an array without first erasing it will produce a 'Redimensioned array' error.

6.4 TYPE CONVERSION FUNCTIONS

Several special functions are provided to enable data to be converted from one type to another. Their argument is of one data type (integer, double-precision number, single-precision number or string), and their result is of another. The most important of these functions are STR$ and VAL, which are used to convert numbers to strings and strings to numbers, respectively.

The argument of **STR$** must be a number or a numeric variable. (It does not matter whether it is an integer or a single- or double-precision number.) The first character of the resultant string is a space if the argument is positive, a minus sign if it is negative. The following characters are the digits of the argument (and the decimal point, if there is one). If the argument is a non-integer variable, the string may have a number of spaces at the end.

Examples

	STR$(8.5)	will return "8.5"
If X=3.14,	STR$(X)	will return "3.14"
If Y%=-7,	STR$(Y)	will return "-7"

Be careful not to confuse STR$ with the function STRING$ (see Section 5.3)

The function **VAL** returns the number at the start of its string argument, if there is one. Any spaces, tabs or line feeds in front of the number are ignored. Plus and minus signs in front of the number are recognized. If, however, there are any other characters at the start of the string, a number following them will not be recognized; if the string contains more than one number, only the first will be returned.

Examples

VAL(" -5.8ABC")	will return	-5.8
VAL("ABC-5.8")	will return	0
VAL("5+4")	will return	5

Numeric data can be converted from one data type to another simply by

assigning it to a variable of the appropriate type. However, there are also some special functions to perform these conversions. **CDBL**, **CINT** and **CSNG** convert their arguments to double-precision, integer and single-precision numbers respectively. Note that CINT can only be used with numbers between –32768 and 32767. These functions are used most often in PRINT statements.

Exercise 6.6

Identify the mistakes in these program segments:

(a)
```
10 DIM A$(20)
20 CLEAR 255
30 A$(15)="Fred Smith"
```

(b)
```
10 DIM A(10,2),B(15)
20 A(10,1)=B(10)
30 DIM B(20)
40 A(10,2)=B(20)
```

(c)
```
10 A$="10.5"
20 B$=VAL(A$)
30 PRINT B$
```

(d)
```
10 CLEAR 300
20 DIM A$(5)
30 A$(0)=STRING$(300,CHR$(255))
```

Exercise 6.7

What result will be returned by:

(a) VAL(" –85,200.7")
(b) STR$(294)
(c) CINT(–876.54) ?

6.5 CONSTANTS

Constants are numbers or strings which do not vary. The types available are strings, integers, single- and double-precision decimal numbers, hexadecimal, octal and binary numbers.

String constants are always enclosed in double quotation marks. If these are omitted, a 'Syntax error' or 'Type mismatch error' will be produced.

Single-precision fixed-point constants, i.e. numbers which contain a decimal point, are distinguished by a trailing exclamation mark. Fixed-

point numbers without a trailing exclamation mark are assumed to be double precision. Double-precision fixed-point constants can be followed by a hash sign, but this is not essential.

Real numbers can also be written in *exponential* form. The letter E is used for single-precision constants, and the letter D for double-precision constants. (The number after the letter represents the power of ten by which the number before the letter has to be multiplied: $nEm = n * 10^m$.)

Hexadecimal, octal and binary numbers are preceded by &H, &O and &B respectively, as shown in Section 6.1.

Examples
String constants:

"New York"
"#12.9"

Single-precision constants:

5.235!
1.3E5

Double-precision constants:

289.54378
6.842D12

Hexadecimal constants:

&HD2
&H3F2A

Octal constants:

&03446
&021

Binary constants:

&B0111000101011
&B1001

7
Operators and Functions

7.1 NUMERIC OPERATORS

Numeric operators are symbols which are used with numeric variables or constants – the *operands* – to form expressions which have a numeric value. There are three different types: arithmetic, relational and Boolean operators.

Arithmetic Operators

All the usual arithmetic operators are available for use on the MSX; these are:

1.	^	exponential	X^Y means X raised to the power Y.
2.	–	negation	Used to denote a negative number.
3.	*	multiplication	
	/	division	
4.	\	integer division	The divisor, dividend and quotient are all rounded down to the nearest integer.
5.	**MOD**	modulus	The remainder of an integer division.
6.	+	addition	
	–	subtraction	

When an expression contains two or more different operators, the operator with the highest precedence is evaluated first. If one operator is used more than once in the expression, then the operators with equal precedence are evaluated from left to right. (This is the way in which normal arithmetic is performed, but unlike the method used by many

pocket calculators, which ignore the rules of precedence and evaluate expressions strictly from left to right.) *Brackets* can be used to alter the order in which calculations are performed. These take precedence over all the operators; their contents are always evaluated first.

Examples
Type:

```
PRINT 13\6/3 <RETURN>
```

You should get an answer of 6. The second operation, 6/3, is performed first; 13\2 produces the result 6.

Type:

```
PRINT 2^ (12.3MOD5)*1.4 <RETURN>
```

The answer should be 5.6. The brackets are evaluated first: 12.3 is rounded down to 12, and the remainder of the integer division of 12 by 5 is 2. The exponential is evaluated next: 2^ 2=4. Then finally, the multiplication gives the result 5.6.

Type:

```
PRINT 43.1*-2+121 <RETURN>
```

The answer is 34.8. The negation is performed first, then the multiplication, then the addition.

Relational Operators

A full range of *relational operators* is also available. These are used to *compare* the values of two constants, variables or expressions, and return a value of -1 if the relation holds true, 0 if it is false. They are:

=	equal to
<>	not equal to
<	less than
>	greater than

<=	less than or equal to
>=	greater than or equal to

Arithmetic operators take precedence over relational operators, so if a relational operator is used in the middle of an expression, the values of the parts of the expression on either side of the relational operator will be evaluated, then compared with one another. It is rare for more than one relational operator to be used in an expression, but when two or more are used, they are evaluated from left to right.

Examples

(a)
```
10 LET A%=3
20 LET B!=2.4
30 PRINT A%=B!
40 PRINT A%<B!
50 PRINT A%>B!
60 END
```

RUN this, and you should get:

```
 0
 0
-1
```

because the relations in lines 30 and 40 are false, and the relation in line 50 is true.

(b) 1.5*3<2*4 will return a value of –1, because the part of the expression to the left of the relational operator has a lower value than the part of the expression to the right (4.5<8).

(c)
```
10 A=1: B=2: C=3
20 ?A<B<C
30 ?C>A<B
40 ?B<C>A
50 ?A<B<A
60 END
```

RUN this, and you should get:

```
-1
```

```
-1
 0
-1
```

In each case, the relation on the left is tested first, and given a value of –1 if true or 0 is false. This value is then compared with the value of the variable on the right. A<B<A will therefore return a value of –1 (true) whenever A is positive, whatever the value of B!

Relational operators are used frequently to *test* the value of program variables, in branching commands (see Chapter 9).

Boolean Operators

These *logical operators* have important uses in machine-code programming, as well as BASIC. They are:

NOT	complement
AND	logical AND
OR	logical OR
XOR	exclusive OR
EQV	equivalent
IMP	implication

They can only be used with integers, so any operand which is not already an integer is automatically converted into an integer. They work by comparing bits of their operands. They are most often used in BASIC in branching instructions, with expressions containing relational operators as their operands. The results obtained can be predicted using this *truth table:*

A	B	NOT A	A AND B	A OR B	A XOR B	A EQV B	A IMP B
true	true	false	true	true	false	true	true
true	false	false	false	true	true	false	false
false	true	true	false	true	true	false	true
false	false	true	false	false	false	true	true

Relational operators take precedence over Boolean operators. Example (c) above showed that illogical results can be obtained when more than one relational operator is used in an expression; the use of AND can eliminate this problem.

Example

```
10 A=1: B=2: C=3
20 ?A<B AND B<C
30 ? C>A AND A<B
40 ?B<C AND C>A
50 ?A<B AND B<A
60 END
```

This program should give the results that you would logically expect:

```
-1
-1
-1
 0
```

Boolean operators can be used for bit manipulation, i.e. for setting or testing individual binary digits. The bits of a binary number are numbered from 0 (the rightmost bit) to 15 (the leftmost bit). This second truth table shows how each bit of the result can be determined from the corresponding bits of the operands:

A	B	NOT A	A AND B	A OR B	A XOR B	A EQV B	A IMP B
1	1	0	1	1	0	1	1
1	0	0	0	1	1	0	0
0	1	1	0	1	1	0	1
0	0	1	0	0	0	1	1

Note this is the same as the first truth table (1=true, 0=false).

NOT simply inverts the bits of its operand, changing 0s to 1s and 1s to 0s.

Example

```
      A=0010110011010101
NOT   A=1101001100101010
```

AND can be used for testing a particular bit. Just AND the number being tested with a number in which only the corresponding bit is 1; the result will be non-zero if the bit being tested is 1, zero if it is 0.

Example

To test bit 4 of a number A, AND A with &B10000.

```
          A=1100101010011001
          B=0000000000010000
A AND B=0000000000010000

          A=0011111110100111
          B=0000000000010000
A AND B=0000000000000000
```

AND can also be used to set particular bits of a number to 0.

Example
To set bits 15 to 12 of a number to 0, AND the number with &B0000111111111111.

```
          A=1011001010110101
          B=0000111111111111
A AND B=0000001010110101
```

OR can be used in a similar way to set particular bits to 1.

Example
To set bits 3 and 7 of a number A, LET A=A OR &B0000000010001000.

```
          A=1100010010100100
          B=0000000010001000
A OR B=1100010010101100
```

This provides a useful way to convert upper-case letters to lower-case and vice-versa. The codes for an upper-case letter and the corresponding lower-case letter are the same apart from bit 5, which is set to 1 in the lower-case code. So to ensure that a character L$ is a lower-case, LET L$=CHR$(ASC(L$) OR 32). To ensure that L$ is an upper-case letter, LET L$=CHR$(ASC(L$) AND -33).

The other three operators, XOR, EQV and IMP, are less important, and can all be derived from combinations of NOT, AND and OR:

```
A XOR B=(A AND (NOT B)) OR ((NOT A) AND B)
A EQV B=(A AND B) OR (( NOT A) AND (NOT B))
A IMP B=(NOT A) OR B
```

7.2 NUMERIC FUNCTIONS

Numeric *functions* are relationships between two sets of numbers, such that each element in one set – the *domain* of the function – is related to a unique element in the second set – the *range*. The *argument* of the function is an element of the domain, and the value returned by the function is an element of the range. For numeric functions, the argument can be a numeric variable, a constant or an expression.

Arithmetic Functions

Two different functions are provided to convert real numbers into integers:

INT rounds the argument down to the nearest whole number
FIX truncates the argument

These produce the same result if the argument is positive, but different results if the argument is negative:

```
INT (3.14)      = 3     FIX (3.14)      = 3
INT (-52.671)   = -53   FIX (-52.671)   = -52
```

The function **ABS** returns the absolute value of the argument:

```
ABS (X) =  X if X is positive
          -X if X is negative
```

The function **SGN** returns the sign: –1 if the argument is negative, 0 if the argument is zero, and 1 if the argument is positive.

SQR is the square root function. Its argument must be positive; the result is calculated to double precision (fourteen significant figures).

Trigonometric Functions

The functions **SIN, COS** and **TAN** require an argument in radians and return its sine, cosine and tangent respectively.

The function **ATN** returns the arc-tangent of the argument; its range is –PI/2 ($-\pi/2$) radians to +PI/2 ($\pi/2$) radians. ATN can be used to give the value of **PI** in double precision, i.e. to fourteen significant figures:

```
PI=4*ATN(1)
```

If you prefer to work in degrees, these can be converted to radians using the formula:

no. of radians = no. of degrees * 180/PI

Natural Logarithms and Exponentials

The functions **LOG** and **EXP** return the logarithm to the base e and the exponential value e^ X of the argument X, respectively.

The argument of LOG must be positive; the argument of EXP must not be greater than 145, or an 'Overflow' error will result.

7.3 RANDOM NUMBERS

A special function, **RND**, is provided to generate random numbers. It is not quite a function in the mathematical sense as it does not return a unique element of its domain (double-precision numbers between 0 and 1) for any one value of its argument. If it did, then the result would clearly not be random! In fact, it is not truly random: the computer is programmed with a fixed sequence of numbers from which the result is obtained, but this sequence is so long that if the starting point is selected randomly the result can be considered to be truly random.

If the argument X is a negative number, RND acts like a 'proper' function and returns a unique result. The negative argument is used to set a pointer to a particular place in the sequence of random numbers, and the function returns the number indicated by this pointer. Thus RND(–1) always returns 0.0438982042082l, RND(–2) always returns 0.9438982042082l, etc. However, if the argument is 0 then the pointer is not moved, and the number which was returned the last time the function was used will be returned again. Try running this program:

```
10 PRINT RND(-1)
20 PRINT RND(0)
30 PRINT RND(-2)
40 PRINT RND(0)
50 PRINT RND(-1)
```

and you should get:

```
0.04389820420821
0.04389820420821
0.94389820420821
0.94389820420821
0.04389820420821
```

The first, second and fifth numbers are the same (the result produced by an argument of –1), and so are the third and fourth numbers (the result produced by an argument of –2).

If the argument of RND is positive, then the pointer is moved to the next number in the sequence. Thus a different result will be obtained each time RND(1) is used, but the numbers will always follow the fixed sequence. To obtain a 'random' sequence of numbers, you can use RND(1) repeatedly, but you must first set the pointer to a random start position in the random number sequence, by using RND with a randomly selected negative argument.

The easiest way to select this first negative argument is to use the **TIME** function, which returns the value of the computer's internal clock. This value depends on the amount of time which has elapsed since the computer was switched on, and so can be expected to be different each time your program is run. The TIME function returns a positive number, so use –TIME as the argument of RND.

To summarize, a sequence of very nearly random double-precision numbers between 0 and 1 can be obtained by using the RND function with arguments –TIME,1,1,1,1,1....

The numbers obtained can be scaled to fall within any range. To produce a random integer within the range *m* to *n*, use the formula:

A%=(1+*n*–*m*)*RND(1)+*m*

Example

```
10 SCREEN 0:WIDTH 30
20 CLS: PRINT"  1   2   3   4   5   6"
30 PRINT"  0   0   0   0   0   0"
40 X=RND(-TIME)
50 N(1)=0:N(2)=0:N(3)=0:N(4)=0:N(5)=0:N(6)=0
60 T%=6*RND(1)+1
70 N(T%)=N(T%)+1
80 LOCATE 5*(T%-1),1:PRINT USING "###";N(T%);
90 GOTO 60
```

Line 10: selects text mode 0, screen width 30 columns.

Line 20: prints column headings.
Line 30: prints a 0 in each column.
Line 40: randomizes RND function.
Line 50: initializes array variables (note that the array need not be dimensioned as it contains less then eleven elements).
Line 60: simulates dice throw (value assigned to T% will be a random integer between 1 and 6).
Line 70: increments appropriate array element.
Line 80: prints new value of incremented array element in appropriate column.
Line 90: returns to line 60.

This program simulates the computer repeatedly throwing a dice. The display shows the number of times each throw has been obtained (the variable N(*n*) contains the number of times the computer has thrown the number *n*, for *n*=1 to 6). Let it run for a few minutes, and check that the numbers in each column remain approximately equal.

7.4 STRING OPERATORS AND FUNCTIONS

Operators

The arithmetic operator + can be used to combine two or more short strings into one long string. This is called *concatenation*. None of the other arithmetic operators can be used with strings.

Example

```
10 A$="THIS IS "
20 B$="A "
30 C$="LONG STRING."
40 A$=A$+B$+C$
50 PRINT A$
60 END
```

Lines 10, 20 and 30: assign character strings to the string variables A$, B$ and C$.
Line 40: forms a new string by concatention of A$, B$ and C$, and assigns this to A$.
Line 50: prints the string assigned to A$.
Line 60: end of program.

RUN this program, and you should get:

```
THIS IS A LONG STRING.
```

When the relational operators are used with strings, they compare the ASCII codes of the characters, starting with the first (leftmost) character of each string. As capital letters have different codes to the corresponding lower-case letters, care must be taken when using these operators to sort strings into alphabetic order. They return the same results as when they are used with numbers: zero if the relation is false, –1 if it holds true. These relations are all true:

```
"1"<"A"
"A"<"AB"
"Z"<"a"
```

Though relational operators can be used to compare two numbers or two strings, they cannot be used to compare a number and a string. If the two items being compared are not of the same type, a 'Type mismatch' error will be produced.

Boolean operators cannot be used with strings, but can be used with the (numeric) results obtained by comparing two strings.

Functions

All these functions have at least one string argument; some have one or more numeric arguments as well. Those which are suffixed by $ produce string results; those without a suffix produce numeric results.

LEN returns the length of a string. All the characters in the string are counted, including spaces and control characters.

The function **INSTR** is used to look for a specified string within a main string. It has three arguments: the position within the main string at which the search is to commence (this is optional and can be omitted, in which case the whole of the main string is searched); the main string; and the specified string. These are separated by commas and enclosed in brackets. If the specified string is found, the function returns the position in the main string of its first character. Otherwise, it returns 0.

Example

```
10 A$="BREAD AND BUTTER"
20 B$="HAM SANDWICH"
```

```
30  C$="BEANS ON TOAST"
40  D$="SWEETS AND CANDY"
50  PRINT INSTR(A$,"AND")
60  PRINT INSTR(B$,"AND")
70  PRINT INSTR(C$,"AND")
80  PRINT INSTR(9,D$,"AND")
90  END
```

Lines 10, 20, 30 and 40: assign strings to A$, B$, C$ and D$.
Lines 50, 60, 70: search for the string "AND" in the strings assigned to A$, B$ and C$, and print the position of its first character.
Line 80: searches characters 9 onwards of the string assigned to D$ for the string "AND", and prints its position.
Line 90: end of program.

RUN this, and you should get:

```
 7
 6
 0
13
```

The first "AND" in D$ was not identified, as the search was started at the ninth character: the N of "AND".

Now edit lines 50, 60, 70 and 80 so that the specified string is followed by a space: "AND ". RUN the program again, and you will get:

```
 7
 0
 0
 0
```

Edit line 10 to read:

```
10  A$="Bread and butter"
```

and you will get:

```
 0
 0
 0
 0
```

The "and" in A$ is no longer identified; capital and lower-case letters are treated as distinct characters within strings.

LEFT$** and **RIGHT$ are very similar. They both have two arguments, a string and a number. LEFT$ returns the given number of characters from the left of the string; RIGHT$ returns characters from the right of the string.

Example

```
10 A$="This is not very easy."
20 L$=LEFT$(A$,8)
30 R$=RIGHT$(A$,5)
40 PRINT L$;R$
50 END
```

Line 10: assigns a string to A$.
Line 20: assigns to L$ the eight leftmost characters of A$.
Line 30: assigns to R$ the five rightmost characters of A$.
Line 40: prints L$ and R$ (on the same line).
Line 50: end of program.

RUN this, and you should get:

```
This is easy.
```

MID$ returns the middle of a string. It has three arguments: the string, the position of the first character to be returned, and the number of characters required. The third argument can be omitted, when all the characters from the first character to the end of the string are returned. This function can also be used back-to-front, to replace part of a string with another string. The length of the string cannot be changed, however; if the replacement string is too long, it will be truncated to fit.

Example

```
10 A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
20 PRINT MID$(A$,13,2)
30 MID$(A$,13,2)="123456789"
40 PRINT A$
50 END
```

Line 10: assigns a string to A$.
Line 20: prints the thirteenth and fourteenth characters of A$.
Line 30: replaces the thirteenth and fourteenth characters of A$ with the

first two characters of the string "123456789".
Line 40: prints A$.
Line 50: end of program.

RUN this, and you should get:

```
MN
ABCDEFGHIJKL12OPQRSTUVWXYZ
```

These string functions can be used together to manipulate strings in a wide variety of ways. One simple application is in adventure games, where a two-word command from the player has to be split up into a verb and a noun:

```
10 INPUT "What do you want to do now";C$
20 S=INSTR(C$," ")
30 V$=LEFT$(C$,S)
40 N$=MID$(C$,S+1)
```

Line 10: prints prompt, inputs string and assigns it to C$ (see Chapter 8).
Line 20: searches for a space in C$.
Line 30: assigns the characters of C$ up to the space (if there is one to V$.
Line 40: assigns the characters of C$ after the space to N$.

This routine assumes the input will consist of two words, separated by a space. In an adventure program, further statements would be included to deal with input which did not take this form.

7.5 SWAP

It is often desirable to sort the numbers or strings stored in the elements of an array into order, so that array element 0 holds the smallest value, element 1 holds the next small value, and so on. The sorting can be accomplished by using relational operators to compare the contents of pairs of elements, then swapping over their contents where necessary. Swapping the contents of two variables normally involves the use of a temporary variable:

```
TEMP=A:A=B:B=TEMP
```

will interchange the values of A and B. There is, however, a special function, **SWAP**, which can be used to interchange the values of two variables without the use of a temporary variable:

```
SWAP <variable 1>,<variable 2>
```

will assign the original value of variable 1 to variable 2, and vice versa. SWAP can be used with numeric variables and string variables, but obviously the two variables must be of the same type.

7.6 USER-DEFINED FUNCTIONS

You can define your own functions, using **DEF FN**. This can be used to create numeric and string functions, and takes the form:

```
DEF  FN<function name>[(<argument>[,<argument>]...)]=
<expression>
```

The function name can be any valid name. If the result of the function is a string, then the name must be suffixed by $. The arguments are the variable names which appear in the expression. You need not have any arguments (the square brackets are used to indicate that the arguments are optional), but you can use up to nine if you wish. The expression which determines the result of the function can contain variables, constants, operators and other functions (including other user-defined functions).

Once a function has been defined, it can be used in the same way as the predefined functions, but its name must be prefixed by FN to indicate to the BASIC interpreter that it is a user-defined function. (If you omit this, you will get a 'Syntax error'.)

Examples

(a)
```
10 DEF FNPOWER(X,Y)=X^ Y
20 A=2:B=3:C=4
30 PRINT FNPOWER(A,B)
40 PRINT FNPOWER(A,C)
50 PRINT FNPOWER(C,1)
60 END
```

Line 10: defines the function FNPOWER, which has two numeric arguments and returns the first argument raised to the power of the second argument.
Line 20: assigns values to A, B and C.
Lines 30, 40 and 50: calculate and print the values of the function FNPOWER with three different pairs of arguments.
Line 60: end of program.

RUN this, and you should get:

```
 8          (2^3)
16          (2^4)
 4          (4^1)
```

(b)
```
10 DEF FNI$(A$)=LEFT$(A$,1)
20 A$="John":B$="Smith"
30 PRINT FNI$(A$);FNI$(B$)
40 END
```

Line 10: defines the string function FNI$, which has one string argument and returns its first character.
Line 20: assigns strings to A$ and B$.
Line 30: calculates and prints FNI$(A$) and FNI$(B$).
Line 40: end of program.

RUN this, and you should get:

```
JS
```

Exercise 7.1

```
10 A=12.5
20 B=0.37
30 A=A-B
40 C%=(A>B)*(A+5*B)-(A<=B)*(A*B MOD2)
50 PRINT C%
```

Work out the value of C% by hand, then RUN the program to check your answer.

Exercise 7.2

Write a program which assigns your full name to the variable N$, then print 'Hello' and your first name only.

Exercise 7.3

```
10 A$="30256725"
20 MID$(A$,3,3)="888"
30 B$=RIGHT$(A$,5)
40 PRINT VAL(B$)
```

Work out what number will be printed, then RUN the program to check your answer.

8

Input Commands and Program Data

8.1 DATA INPUT

Almost all programs need some input from the user. Mathematical programs need numbers to be fed in, word-processing programs need text, arcade games need directions to move and fire. MSX BASIC provides a range of different input commands to deal with different situations.

The command **INPUT** can be used for inputting numeric data or character strings. The syntax for this command is:

INPUT ["<prompt>";]<variable>[,<variable>, . .]

The data is entered from the keyboard, the <RETURN> key being pressed to indicate when the entry has been concluded. If more than one data item is being entered, the items may be separated by commas or by <RETURN>. The items are assigned to the variables specified in the INPUT statement, which must, of course, be of the right type; if a string is entered when a numeric variable is specified, the input will be rejected.

When the computer encounters an INPUT statement in a program, it prints the prompt (if one has been specified) and a question mark, then waits for the the required number of data items to be entered. The prompt can be used to indicate to the user what type of input is required.

As commas are used to separate data items, problems will arise if you try to input a single string variable which includes a comma: the comma and any characters after it will be ignored. To input string data which includes commas another command, **LINE INPUT**, must be used. This command can incorporate a prompt, but does not automatically print a question mark. It cannot be used for numeric data. The syntax is:

LINE INPUT ["<prompt>";]<string variable>

Examples

(a)
```
10 INPUT "What is your name";A$
20 INPUT "How old are you";B
30 PRINT A$; "is";B
```

RUN this, and enter the input shown in bold type:

What is your name? **Mary Brown <RETURN>**
How old are you? **Twenty <RETURN>**
?Redo from start
How old are you? **20 <RETURN>**
Mary Brown is 20

The computer rejected the input 'Twenty', as this is a character string and could not be assigned to the numeric variable B. The prompt was repeated and the revised input, the number 20, was accepted.

(b)
```
10 INPUT "First number";A
20 INPUT "Second number";B
30 PRINT "The sum of the numbers is";A+B
40 GOTO 10
```

This little program will ask you for pairs of numbers and add them together for you. It will repeat itself indefinitely, so when you are fed up with entering numbers, press <CTRL> and <STOP>.

(c)
```
10 INPUT "Name";N$
20 LINE INPUT "Address?";A$
30 INPUT "Telephone no.";T
```

The INPUT command is used for the name and telephone number; LINE INPUT is used for the address, as this may include commas. Note that the prompt 'Address?' includes a question mark, as this is not printed automatically with LINE INPUT.

With the INPUT and LINE INPUT commands, the <RETURN> key must be pressed to tell the computer when the input has been completed. This is not necessary with the **INPUT$** command, which inputs a specified number of characters. It takes the form:

<string variable> = INPUT$<no. of characters>

INPUT$ does not produce a question mark and cannot incorporate a prompt, but you can precede it with an appropriate PRINT command if a prompt and/or question mark is required. The input is not automatically displayed on the screen. The data must again be assigned to a string variable. INPUT$ is very useful if you know in advance how many characters are required, e.g. if a five-character name is to be entered for inclusion in a high-score table at the end of a game, or if you want a Y (yes) or N (no) answer to a question.

INPUT$ (1) can be used to 'hold' a display on the screen until a key is pressed: the display might be the instructions for a game. Just end your instructions routine with the line:

```
PRINT"Press any key to continue":A$=INPUT$(1)
```

The character which is input here is irrelevant; you can assign it to any string variable which is not being used for any other purpose.

With INPUT$, as with INPUT, the computer stops and waits for the data to be entered. When the **INKEY$** command is used, the computer does not wait for input, but simply checks to see if a key has been pressed, then carries straight on to the next instruction. This can be very useful in games programs, where you do not want to wait until the player gives an instruction to move or to fire, but simply to see if the instruction has been given. Here again, nothing is automatically printed on the screen. The syntax for INKEY$ is:

```
<string variable>=INKEY$
```

If a key has been pressed, then the appropriate character is assigned to the specified variable. If no key has been pressed, then an empty string ("") is assigned to this variable.

After an input command, you will often want the computer to test the input to see if it satisfies certain conditions, then to base its next action on the results of this test. The simplest way to do this is to use the **IF ... THEN ... ELSE** command, which takes this form:

```
IF <condition> THEN <statement> [ELSE <statement>]
```

IF is followed by a condition which may be true (non-zero) or false (zero). If the condition is true, the computer executes the statement following the word THEN. If the condition is false the computer executes the statement following the word ELSE, or if ELSE is omitted, goes straight on to the next program line. This command is described in more detail in Chapter 9.

Example
This program uses both INPUT$ and INKEY$:

```
 10 SCREEN 0: WIDTH 40
 20 ?"Do you want instructions?"
 30 A$=INPUT$(1)
 40 IF A$="N" OR A$="n" THEN GOTO 200
 50 REM Instructions
 60 CLS:PRINT TAB (12) "Instructions"
 70 ?"A small square will move across the screen."
 80 ?"Press the "L" key to make it move from right to left."
 90 ?"Press the "R" key to make it move from left to right."
100 ?"Press any key to start."
110 B$=INPUT$(1)
190 REM Initialization
200 CLS
210 X=20:D=1
220 REM Start of main program routine
230 LOCATE X,10
240 PRINT CHR$(219)
250 C$=INKEY$
260 IF C$="R" OR C$="r" THEN D=1
270 IF C$="L" OR C$="1" THEN D=-1
280 X=X+D
290 F X<0 THEN X=39
300 IF X>39 THEN X=0
310 CLS
320 GOTO 230
```

Line 10: selects text mode 0, screen width 40 columns.
Line 20: prints prompt.
Line 30: waits for a single character to be entered, and assigns this character to A$.
Line 40: jumps to start of initialization routine if the N key (SHIFTed or unSHIFTed) was pressed.
Line 50: comment.
Line 60: clears the screen, prints 'Instructions' in the middle of the top row.
Lines 70, 80, 90: print instructions.
Line 100: prints prompt.
Line 110: waits for a keypress.
Line 190: comment.

Line 200: clears the screen.
Line 210: sets the initial X co-ordinate, at which a square is to be printed, and the initial value of D (the amount by which X is to be incremented).
Line 220: comment.
Line 230: moves the cursor to column X, row 10.
Line 240: prints a small square at the cursor position.
Line 250: checks to see if a key has been pressed.
Line 260: if the R key has been pressed, sets the value of D to 1.
Line 270: if the L key has been pressed, sets the value of D to –1.
Line 280: increments X by D.
Line 290: checks to see if the X co-ordinate is off the left-hand edge of the screen, and if so, resets it to the right-hand edge.
Line 300: checks to see if the X co-ordinate is off the right-hand edge of the screen, and if so, resets it to the left-hand edge.
Line 310: clears the screen.
Line 320: returns to line 230 (to print the square in a new position).

The instructions sequence (lines 70–90) explains what this program does.

8.2 PUTTING DATA IN A PROGRAM

Your program data need not all be entered from the keyboard; some of it can be put in the program itself, in **DATA** statements. The **READ** command can then be used to assign this data to variables.

DATA statements take the form:

DATA <data item>,<data item>, ...

A single DATA statement can contain as many data items as can be fitted into a single program line (a maximum of 255 characters). The items can be either numbers or strings.

READ commands take the form:

READ <variable>,<variable>, ...

DATA statements can be put anywhere in the program, as the computer does not execute them; it simply stores the data until it is required. The READ commands read the data in the order in which it

appears in the program, i.e. the data items in the DATA statement with the smallest line number are read first. The data items must be assigned to variables of the right type: string data must be read into string variables, and numeric data must be read into numeric variables.

When the data items have all been read, any attempt to READ another item will produce an 'Out of DATA' error. However, you can **RESTORE** data so that it can be read again. RESTORE on its own restores all the data in the program; RESTORE <line no.> restores the data in a particular program line, so that this data will be read first.

If you want to read a number of different data items into a variable in turn, it is a good idea to put a 'dummy' data item at the end of the last DATA statement. When each item is READ, an IF ... THEN command can be used to check whether this item has been reached and, if so, end the program (or go on to the next routine). If you are using a single READ statement to read several different data items, you will of course have to use more than one 'dummy' item to prevent an 'Out of data' error.

Example

```
 10 REM Initialization
 20 DIM A(5)
 30 READ A(1),A(2),A(3),A(4),A(5)
    .
    .
100 REM Start of main program routine
100 READ M$,N$,X,Y
    .
    .
200 RESTORE 310
210 GOTO 100
300 DATA 1,2,3,4,5
310 "England", "U.S.A.",45,923
```

Line 10: comment.
Line 20: dimensions array A.
Line 30: reads first five data items into A(1) – A(5).
Line 100: comment.
Line 110: reads next four data items into M$, N$,X, Y.
Line 200: restores the data in line 310, so it can be READ again.
Line 210: returns to line 100.
Line 300: data to be READ by line 30.
Line 310: data to be READ by line 110.

The data in line 300 is used in the initialization routine at the start of this outline program, and is not needed again. The data in line 310 is read each time the main program routine is executed, so this line is restored at the end of the main routine.

Example

```
10 REM Shopping List
20 READ A$:IF A$="END" THEN END
30  PRINT "How many ";A$;:INPUT N
40  IF N>0 THEN LPRINT A$ TAB(50)N
50  GOTO 20
60  DATA Soap, Toothpaste, Shampoo,END
```

Line 10: comment.
Line 20: reads next data item and checks to see if end of list has been reached.
Line 30: prints prompt, inputs number of item required.
Line 40: prints item and number required on printer.
Line 50: returns to line 20 to read next data item.
Line 60: shopping list data, finishing with dummy item 'END'.

If you have a printer, you can use this short program to print out a shopping list for you each week. Simply list all the items you may want to buy in DATA statements at the end of the program, remembering to put the dummy item 'END' at the end of your list.

Exercise 8.1

Write a routine which will print a five-letter name in the centre of the screen as it is entered on the keyboard.

Exercise 8.2

Write a routine which will print this question from a multiple-choice test, input the user's answer, then print 'Well done!' if the answer is correct, or 'You had better re-read this chapter' if it is incorrect:

Question 5. Which of these commands automatically prints a question mark?
(1) INPUT
(2) LINE INPUT
(3) INPUT$
(4) INKEY$

Answer 1, 2, 3 or 4?

Exercise 8.3

Write a routine which will PRINT CHR$(205) at (10,10), CHR$(207) at (6,14), CHR$(208) at (14,14) and CHR$(206) at (10,18), putting the co-ordinates and character codes in a DATA statement. (Put dummy data items at the end of the DATA statement, to mark the end of the data.)

9
Loops and Branches

9.1 GOING AROUND IN CIRCLES

It is often necessary to repeat an operation or series of operations a number of times within a program. The values of the variables used may change each time the procedure is repeated, but the procedure itself does not. It is of course possible to repeat the necessary commands several times, but this is both long winded and very wasteful of memory space; it is obviously better if the program can be written so that a single set of commands is executed the required number of times.

One option is to use a subroutine. These were introduced in Chapter 4, and are invaluable if a procedure is to be carried out at several different places within the program. However, if the procedure is repeated several times in succession, it is neater to construct a program loop than to use repeated calls to a subroutine.

A loop can be constructed by setting up a counter, which is incremented each time the procedure is repeated. At the end of the procedure the value of the counter is tested, and if it has not yet reached the required value, control is returned to the start of the procedure using a GOTO command.

This technique can be used to print out a multiplication table:

```
10 REM Multiplication Tables Version 1
20 INPUT "Which table would you like"; N%
30 C%=1
40 PRINT C%;" × ";N%;" = ";C%*N%
50 C%=C%+1
60 IF C%<13 THEN GOTO 40
70 END
```

Line 10: comment.

Line 20: prints prompt, inputs number of table required (N%).
Line 30: initializes counter.
Line 40: prints next line of table.
Line 50: increments counter.
Line 60: checks to see if counter has reached end value; if not, returns to line 40.
Line 70: end of program.

A special BASIC command can be used to set up a loop like this: the **FOR...NEXT** command. The format is:

```
FOR <counter> = <start value> TO <end value>[STEP <increment>]
.
.
NEXT [<counter>]
```

The counter can be any numeric variable. It may be used as an ordinary variable within the loop, but be careful not to include any commands which will change its value or the procedure may be repeated more or fewer times than you expected!

The start value, end value and increment may be positive or negative integers or real numbers. The counter is incremented at the end of the loop, and the loop is repeated if the end value has not been exceeded: a loop starting with FOR C=1 TO 10 STEP 2 will be executed five times, with C taking the values, 1, 3, 5, 7 and 9. If the end value is greater than the start value and the increment is negative, or the end value is less than the start value and the increment is positive, the loop will be executed only once. STEP and the increment may be omitted (hence the square brackets), when an increment of 1 will be assumed.

Loops can be nested inside one another, but must not overlap, i.e. the last loop to be opened with a FOR command must be the first to be closed with a NEXT command. The counter variable may be omitted after the word NEXT, when the counter for the last loop to be opened will be assumed. If the computer encounters a NEXT command without a preceding FOR command, it will stop with a 'NEXT without FOR' error. This will also happen if you GOTO the middle of a loop!

Using a FOR...NEXT loop makes the multiplication tables program shorter, as well as making its structure clearer:

```
10 REM Multiplication Tables Version 2
20 INPUT "Which table would you like";N%
```

```
30  FOR C%=1 TO 12 STEP 1
40 PRINT C%;" × ";N%;" = ";C%*N%
50 NEXT C%
60 END
```

Line 10: comment.
Line 20: prints prompt, inputs number of table required.
Line 30: sets up loop to be executed for values of C from 1 in increments of 1 to 12.
Line 40: prints next line of table.
Line 50: end of loop.
Line 60: end of program.

An empty loop, where the FOR command is followed immediately by the NEXT command, can be used to produce a pause. The length of the pause will obviously depend on the parameters used; FOR D=0 TO 1000:NEXT D produces a pause of about two seconds.

Example

```
10  REM CALCULATE A MEAN
20  INPUT "How many numbers";N%
30  T=0
40  FOR C=1 TO N%
50  INPUT "Number";A
60  T=T+A
70  NEXT C
80  M=T/N%
90  PRINT "The mean is ";M
```

Line 10: comment.
Line 20: prints prompt, inputs number of numbers to be averaged (N%).
Line 30: initializes total (T).
Line 40: sets up loop, to be executed for values of C from 1 in increments of 1 to N%.
Line 50: prints prompt, inputs next number.
Line 60: adds number to total.
Line 70: end of loop.
Line 80: calculates mean of the numbers.
Line 90: prints mean.

Here a numeric variable, rather than a constant, determines the end

value of the counter.

Example

```
 10 REM SORT ROUTINE
 20 INPUT "How many items";N%
 30 DIM A$(N%-1)
 40 INPUT "First item";A$(0)
 50 FOR C1%=1 TO N%-1
 60 INPUT "Next item";A$(C1%)
 70 NEXT C1%
 80 FOR C2%=N%-2 TO 0 STEP -1
 90 FOR C3%=0 TO C2%
100 IF A$(C3%)>A$(C3%+1) THEN SWAP A$(C3%),A$(C3%+1)
110 NEXT C3%
120 NEXT C2%
130 FOR C4%=0 TO N%-1
140 PRINT A$(C4%)
150 NEXT C4%
160 END
```

Line 10: comment.
Line 20: prints prompt, inputs number of items to be sorted into ascending order.
Line 30: dimensions array to contain items to be sorted.
Line 40: prints prompt, inputs item and assigns it to A$(0).
Line 50: sets up loop, to be executed for values of C1% from 1 in increments of 1 to N%-1.
Line 60: prints prompt, inputs next item and assigns it to A$(C1%).
Line 70: end of loop.
Line 80: sets up outer loop, to be executed for values of C2% from (N%-2) in increments of -1 to 0.
Line 90: sets up inner (nested) loop, to be executed for values of C3% from 0 in increments of 1 to C2%.
Line 100: compares a pair of variables, interchanges their values if necessary.
Line 110: end of inner loop.
Line 120: end of outer loop.
Line 130: sets up loop to be executed for values of C4% from 0 in increments of 1 to N%-1.
Line 140: prints next item.
Line 150: end of loop.
Line 160: end of program.

This routine uses four different loops, two of which are nested, to input a selection of character strings, sort them into ascending order (numbers first, then capital letters, then lower-case letters), then print them out. There are a lot of different methods which can be used for sorting items; this is one of the slowest, but the easiest to understand.

9.2 BRANCHES

Branching commands enable you to make the computer follow one of two or more different paths through a program, depending on the value of a variable or expression. The **IF...THEN...ELSE** command was introduced in Chapter 8; IF is followed by a condition, which the computer tests to determine whether it is true or false. If the condition is true, the computer obeys the commands following the word THEN. If it is false, the computer obeys the commands following the word ELSE, or if ELSE is omitted, continues to the next line of the program.

THEN and ELSE are very often followed by a GOTO command. An abbreviated form can be used for this: THEN GOTO can be replaced by just THEN or just GOTO, and ELSE GOTO can be replaced by just ELSE. For example,

```
IF A=1 THEN GOTO 200 ELSE GOTO 300
```

can be abbreviated to

```
IF A=1 THEN 200 ELSE 300
```

or

```
IF A=1 GOTO 200 ELSE 300
```

IF ... THEN ... ELSE commands can be nested, by following the THEN or the ELSE with another IF...THEN...ELSE. As the ELSE part of the command is optional, there may be fewer ELSEs than THENs, in which case each ELSE will be assumed to relate to the nearest THEN. Nested commands can be rather confusing; it may be better to replace the nested command with GOTO another program line.

A single IF...THEN...ELSE command causes the program to branch in two different directions. There may be times when you want

more than two different branches – if, for example, you have set up a menu of options and you want the program to branch to a different routine depending on which option is selected. This can be accomplished by using multiple or nested IF...THEN...ELSE commands, but there is also a special command for this purpose, the **ON...GOTO** command. It takes the form:

ON <variable or expression> GOTO <line no.>,<line no.>,<line no.>,...

The computer evaluates the variable or expression given; if its value is 1, it will GOTO the first line number, if the value is 2 it will GOTO the second line number, and so on. If the value is zero or greater than the number of line numbers given, it will simply go on to the next line of the program.

The **ON ... GOSUB** command is identical to ON ... GOTO, except that it transfers execution to subroutines rather than to individual program lines. You must, of course, end each of the subroutines with a RETURN command.

Example

Assume that the variables A and B can each take the value 0, 1 or 2. These routines both set up nine branches, one for each combination of values of A and B.

(a)
```
100 IF A=0 THEN IF B=0 GOTO 200 ELSE IF B=1 GOTO
300 ELSE 400
110 IF A=1 THEN IF B=0 GOTO 500 ELSE IF B=1 GOTO
600 ELSE 700
120 IF B=0 GOTO 800 ELSE IF B=1 GOTO 900 ELSE 1000
```

(b)
```
100 ON A+1 GOTO 110,120,130
110 ON B+1 GOTO 200,300,400
120 ON B+1 GOTO 500,600,700
130 ON B+1 GOTO 800,900,1000
```

Example

```
10  CLS
20  ?"(1) First option"
30  ?"(2) Second option"
40  ?"(3) Third option"
50  ?"(4) Fourth option"
60  INPUT "Select option 1,2,3 or 4";A%
70  ON A% GOSUB 100,200,300,400
```

```
 80 GOTO 10
100 REM Subroutine for first option starts here
    .
    .
190 RETURN
200 REM Subroutine for second option starts here
    .
    .
290 RETURN
300 REM Subroutine for third option starts here
    .
    .
390 RETURN
400 REM Subroutine for fourth option starts here
    .
    .
490 RETURN
```

Line 10: clears the screen.
Lines 20–50: print menu of four options.
Line 60: prints prompt, inputs option number.
Line 70: branches to subroutine starting at line 100 for option 1, subroutine 200 for option 2, subroutine 300 for option 3, or subroutine 400 for option 4.
Line 80: returns to line 10 (note that the subroutines all RETURN execution to this line).
Lines 100–190: subroutine for option 1.
Lines 200–290: subroutine for option 2.
Lines 300–390: subroutine for option 3.
Lines 400–490: subroutine for option 4.

This outline program prints a menu of options on the screen. After an option has been selected, the ON...GOSUB command in line 70 directs the computer to the appropriate subroutine. When the end of the subroutine is reached, execution is RETURNed to line 80, and thence back to the beginning of the program. Line 80 also sends the computer back to the beginning if an invalid option is selected.

Exercise 9.1

Write a program using a FOR...NEXT loop which will print all the available graphics characters on the mode 1 text screen.

Exercise 9.2

The input routine for an adventure program, starting at line 100, must accept the input of a single letter direction (N, S, E or W), in which case execution will be transferred to a movement routine starting at line 500, or input consisting of two words separated by a space, in which case execution will be transferred to a routine starting at line 700. If the input takes any other form, the message 'I don't understand that' must be printed. Write the routine.

10
Graphics

10.1 THE GRAPHICS MODES

The **SCREEN** command, used to switch between the two text modes, can also be used to access two different graphics modes, mode 2 and mode 3. These modes can only be used in programs; when the computer has finished running a graphics program, it returns to one of the text modes to give the 'OK' prompt. Thus if you want your graphics to stay on the screen, you must put a GOTO command at the end of your program to make the computer 'stick', or use an input command so that the program will not end until a key has been pressed, i.e. end the program with:

```
1000 GOTO 1000
```

or

```
1000 A$=INPUT$(1):END
```

Graphics mode 2, which is selected with the command SCREEN 2, is a high-resolution mode. Text mode 1 provides 32 columns and 24 rows of character positions, each consisting of an 8×8 block of pixels, i.e. there are 256×196 pixels on the screen. In mode 2 the screen is arranged in the same way, but the co-ordinates refer to individual pixels rather than character positions; so the X co-ordinates run from 0 (at the left of the screen) to 255, and the Y co-ordinates run from 0 (at the top of the screen) to 191. Each of the eight rows of eight pixels in each character position is assigned two bytes of the video memory. One of these bytes determines the background and foreground colours for that row of eight pixels, and the other byte determines whether the individual pixels are in the background colour (the corresponding bit is

set to 0) or the foreground colour (the bit is set to 1). Thus all the sixteen available colours can be used on the screen at once, but only two colours can be used in each group of eight pixels. The graphics resolution is 256×192 and the colour resolution is 32×192.

Graphics mode 3, the multicolour mode, is selected with the command SCREEN 3. This mode uses the same co-ordinate system as mode 2, but here the character positions are each divided into four 4×4 blocks of pixels, which can be coloured individually. Two bytes of the video memory are used to determine the colours of the four blocks of pixels in each character position. Pictures drawn in this mode have a noticeably chunky look to them.

The same BASIC commands are used to produce pictures in both these graphics modes, though the results obtained from them obviously differ, depending on the mode which is being used. All the commands use the co-ordinate system described above. Figure 4 shows this co-ordinate system.

No cursor appears on the graphics screen, but the computer uses an invisible cursor to keep track of its position on the screen. The cursor position is updated whenever a graphics command is used. Most graphics commands can specify a point using either absolute co-ordinates, i.e. the normal X and Y co-ordinates, or relative co-ordinates, i.e. co-ordinates relative to the position of the graphics cursor (using the graphics cursor position as the origin). The relative co-ordinates of a point can be calculated by subtracting the graphics cursor co-ordinates from the absolute co-ordinates – but they are usually used to avoid the necessity for calculating the absolute co-ordinates. They are particularly useful in graphics subroutines. If only relative co-ordinates are used in a subroutine, the same design can be produced in

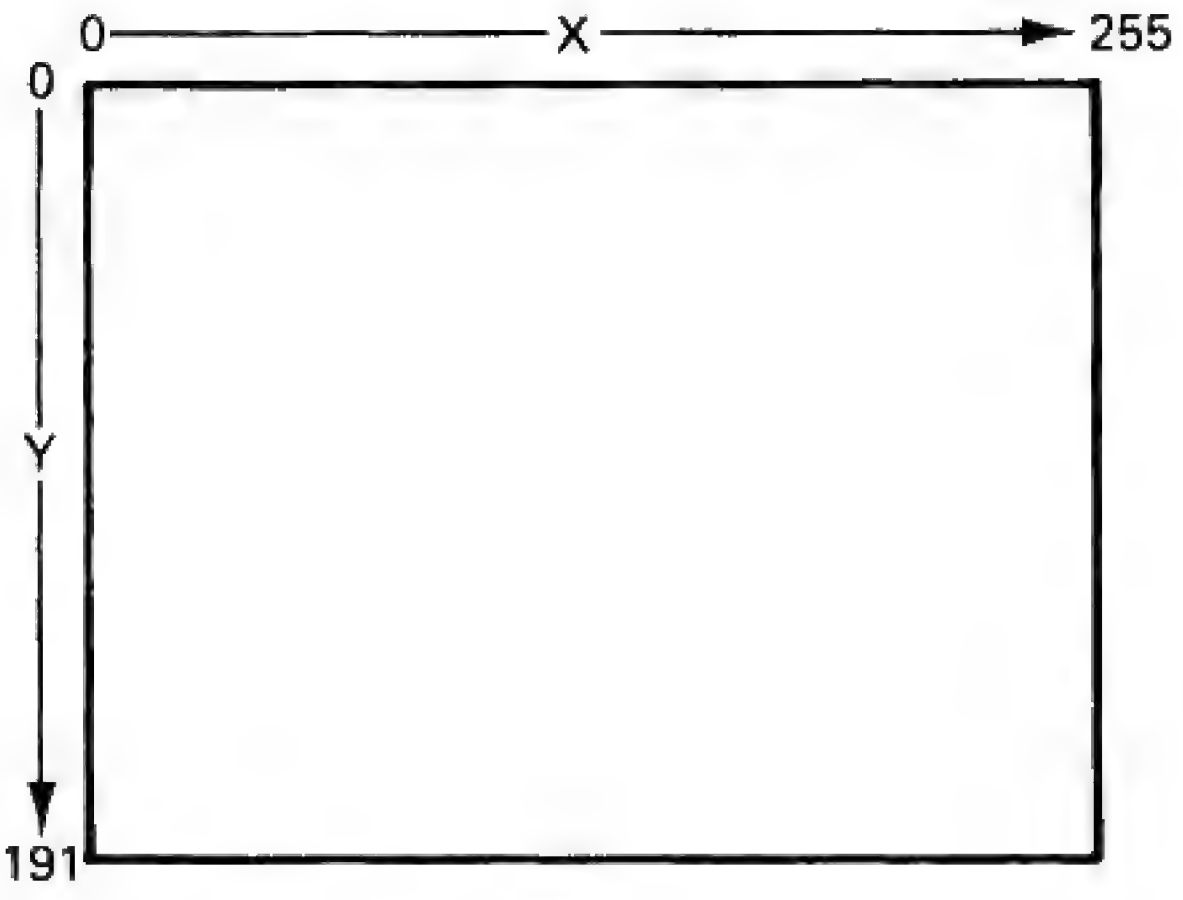


Figure 4 *The graphics co-ordinates*

more than one position on the screen simply by moving the graphics cursor to each start location in turn, then calling the subroutine. The word **STEP** is used to identify relative co-ordinates:

(A,B) refers to the point with X co-ordinate A and Y co-ordinate B.
STEP (A,B) refers to the point A pixels right and B pixels down from the current cursor position.

In mode 2, when a command is used which changes the foreground colour for a group of pixels, any pixels in that group which were displayed in a previous foreground colour will automatically change to the new foreground colour. This means that if you try to use more than two colours in any one group, the picture will 'smudge'. Careful planning is necessary if you want to draw detailed multicoloured pictures in this mode.

In mode 3, changing the colour of any one pixel in a block will automatically change the colour of the whole block.

It is advisable to design your graphics on paper before trying to produce them on screen. Sample worksheets are shown in Appendix C. You can photocopy these, or make your own.

10.2 COLOURS

The MSX colour set consists of sixteen different colours, including black, white and transparent. They are numbered as follows:

0 transparent
1 black
2 medium green
3 light green
4 dark blue
5 light blue
6 dark red
7 cyan/sky blue
8 medium red
9 light red
10 dark yellow
11 light yellow
12 dark green
13 magenta

14 grey
15 white

The **COLOR** command (note the American spelling) can be used with these colour numbers to set the screen foreground, background and border colours. The syntax used is:

COLOR [<foreground colour>],[<background colour>][,<border colour>]

When this command is used in one of the text modes, the screen colours change immediately. When it is used in one of the graphics modes, nothing appears to happen. The computer keeps a record of the current colours, however, and when a command which depends on the screen colours is executed, these recorded colours are used. Thus if a COLOR command is followed by CLS, the graphics screen will be cleared to the specified background and border colours.

Example

```
10 CLS
20 FOR C=0 TO 15
30 COLOR 15,C,C
40 FOR DELAY=0 TO 1000:NEXT DELAY
50 NEXT C
60 COLOR 15,4,4
70 END
```

Line 10: clears the screen.
Line 20: sets up loop to be executed for values of C from 0 to 15 (i.e. for each colour code).
Line 30: sets the background and border to colour C.
Line 40: delay loop.
Line 50: end of main loop.
Line 60: selects the default colour set.
Line 70: end of program.

This program will display all the available colours in turn (in the current text mode). The delay loop (line 40) is put in to slow the program down enough for you to see the colours.

When the foreground colour is set to transparent, the background colour shows through; when the background is set to transparent, the

border shows through. If the border is set to transparent, it appears black.

10.3 POINTS

The commands **PSET** and **PRESET** are used to set a point on the graphics screen to a specified colour, or to the current foreground colour (PSET) or the current background colour (PRESET). The syntax for these commands is:

PSET (<[X co-ordinate>,<Y co-ordinate>)[,<colour no.>]
PRESET (<X co-ordinate>,<Y co-ordinate>)[,<colour no.>]

Note that when a colour is specified, the effects of PSET and PRESET are identical.

The X and Y co-ordinates can be specified by constants, variables or expressions, using absolute or relative co-ordinates. They must be within the integer range (–32768 to 32767), or an error message will be produced. They must, of course, be within the screen range (0 to 255 for X, 0 to 191 for Y) if you want the command to have any effect! The graphics cursor is moved to the specified point.

Examples
(a) This program displays four yellow dots on a dark green background:

```
10 SCREEN 2
20 COLOR 11,12,12:CLS
30 PSET (50,50)
40 PSET STEP (10,0)
50 PSET STEP (0,10)
60 PSET STEP (-10,0)
70 GOTO 70
```

Line 10: selects graphics mode 2.
Line 20: sets the foreground to light yellow, the background and border to dark green, and clears the screen.
Line 30: sets the point (50,50) to yellow.
Line 40: sets the point (60,50) to yellow.

Line 50: sets the point (60,60) to yellow.
Line 60: sets the point (50,60) to yellow.
Line 70: holds the display on the screen.

(b)
```
10 SCREEN 3
20 COLOR 1,15,15:CLS
30 FOR X=32 TO 220 STEP 4
40 PSET (X,X-30)
50 NEXT X
60 GOTO 60
```

Line 10: selects graphics mode 3.
Line 20: sets foreground to black, background and border to white, and clears the screen.
Line 30: sets up a loop to be executed for values of X from 32, in increments of 4, to 220
Line 40: sets the point (X,X–30) to black.
Line 50: end of loop.
Line 60: holds the display on the screen.

A diagonal line of black 4×4 pixel squares appears on a white screen.

The function **POINT** returns the colour code of a specified point. If the point specified is off the screen, it returns –1. The syntax used is:

POINT (<X co-ordinate>,<Y co-ordinate>)

Example

```
10 SCREEN 2:COLOR 7,8,9
20 PSET (10,10):PSET (20,20),10:PRESET(30,30)
30 P(1)=POINT (10,10):P(2)=POINT (20,20)
40 P (3)=POINT (30,30):P (4)=POINT (–10,–10)
50 SCREEN 0
60 PRINT P(1);P(2);P(3);P(4)
70 END
```

Line 10: selects graphics mode 2, foreground sky blue, background medium red, border light red.
Line 20: sets the point (10,10) to the foreground colour (sky blue), the point (20,20) to colour 10 (dark yellow) and the point (30,30) to the background colour (medium red).
Line 30: assigns the colour code of point (10,10) to the array variable P(1) and the colour code of (20,20) to P(2).
Line 40: assigns the colour code of (30,30) to P(3) and the colour code

of (–10,–10) (which is off the screen) to P(4).
Line 50: selects text mode 0.
Line 60: prints the values of P(1),P(2),P(3) and P(4).
Line 70: end of program.

When you RUN this program the numbers printed out should be 7, 10, 8 and –1. (Note that they cannot be printed on the graphics screen; you must switch back to one of the text screens before PRINTing.)

10.4 LINES AND BOXES

The **LINE** command is used, as its name suggests, to draw lines. The co-ordinates of the end of the line must be specified; absolute or relative co-ordinates may be used. The co-ordinates of the start of the line may also be specified, the current position of the graphics cursor being used if they are omitted. The graphics cursor moves to the end of the line. The syntax is:

LINE [(<start co-ordinates>)]-(<end co-ordinates>)[,<colour>[,B] [F]]

If no colour is specified, the current foreground colour is used. The significance of the optional parameters B and F will be explained later.

Examples

(a)
```
10 SCREEN 2:COLOR 9,6,6:CLS
20 LINE(10,10)-(240,10)
30 LINE-(240,180)
40 LINE-STEP(-230,0)
50 LINE (10,180)-STEP (0,-170)
60 GOTO 60
```

Line 10: selects graphics mode 2, foreground light red, background and border dark red and clears the screen.
Line 20: draws a light red line from (10,10) to (240,10).
Line 30: draws a light red line from (240,10) to (240,180).
Line 40: draws a light red line from (240,180) to (10,180).
Line 50: draws a light red line from (10,180) to (10,10).
Line 60: holds the display on the screen.

(b)
```
10 SCREEN 3:COLOR 11,13,11:CLS
20 LINE (20,20)–(50,100),1
30 GOTO 30
```

Line 10: selects graphics mode 3, foreground and border light yellow, background magenta and clears the screen.
Line 20: draws a line in colour 1 (black) from (20,20) to (50,100).
Line 30: holds the display on the screen.

(c)
```
10 SCREEN 2:COLOR15,1,1:CLS
20 FOR C=2 TO 15
30 LINE(0,C*8)-(255,C*8),C
40 NEXT C
50 GOTO 50
```

Line 10: selects graphics mode 2, foreground white, background and border black and, clears the screen.
Line 20: sets up a loop to be executed for values of C from 2 to 15.
Line 30: draws a line in colour C from (0,C*8) to (255,C*8).
Line 40: end of loop.
Line 50: holds the display on the screen.

Fourteen differently coloured horizontal lines are drawn on a black background.

(d)
```
10 SCREEN 2:COLOR15,1,1:CLS
20 FOR C=2 TO 15
30 LINE(C*4,0)-(C*4,191),C
40 NEXT C
50 GOTO 50
```

Line 10: selects graphics mode 2, foreground white, background and border black, and clears the screen.
Line 20: sets up a loop to be executed for values of C from 2 to 15.
Line 30: draws a line in colour C from (C*4,0) to (C*4,191).
Line 40: end of loop.
Line 50: holds the display on the screen.

Fourteen vertical lines are drawn – but they are not all different colours! Because of the limited colour resolution in mode 2, the colour of alternate lines is changed to the colour of the line to their right; the result is two lines of each of seven colours.

Horizontal and vertical lines drawn with the LINE command are always straight, but lines drawn at an angle have a jagged appearance, because they are made up of discrete pixels. If a number of lines are drawn very close together, the jagged effect can produce very attractive patterns. This program creates a patterned border at the edge of the screen:

```
10 SCREEN 2:COLOR 11,6,6
20 FOR X=0 TO 254 STEP 2
30 LINE(X,0)-(254-X,190):NEXT X
40 FOR Y=0 TO 190 STEP 2
50 LINE(0,Y)-(254, 190-Y):NEXT Y
60 GOTO 60
```

Line 10: selects graphics mode 2, foreground light yellow, background and border dark red.
Line 20: sets up a loop to be executed for values of X from 0 in increments of 2 to 254.
Line 30: draws a line in yellow from (X,0) to (254–X,190); endof loop.
Line 40: sets up a loop to be executed for values of Y from 0 in increments of 2 to 190.
Line 50: draws a line in yellow from (0,Y) to (254,190 – Y); end of loop.
Line 60: holds the display on the screen.

If the colour code at the end of the LINE command is followed by a comma and the letter B (for box), an outline rectangle will be drawn instead of a line. The sides of the rectangle are parallel to the edges of the screen, and the co-ordinates specified are those of the top left-hand and bottom right-hand corners respectively. (If no colour code is specified, the letter B must be preceded by two commas.)

If the letter B is followed by an F (for filled), a solid rectangle of the specified colour will be produced instead of an outline rectangle.

Example

```
10 SCREEN 2:COLOR 1,14,14:CLS
20 R%=RND(-TIME)
30 FOR A%=88 TO 8 STEP -8
40 C%=15*RND(1)+1
50 LINE (120-A%,96-A%)-(120+A%,96+A%),C%,BF
60 NEXT A%
70 GOTO 70
```

Line 10: selects graphics mode 2, foreground black, background and border grey, and clears the screen.
Line 20: randomizes the RND function.
Line 30: sets up a loop to be executed for values of A% from 88 in decrements of 8 to 8.
Line 40: selects a random integer, C%, between 1 and 15.
Line 50: draws a solid rectangle in colour C% with top left-hand corner 120–A%,96–A%) and bottom right-hand corner (120+A%,96+A%).
Line 60: end of loop.
Line 70: holds the display on the screen.
A series of randomly coloured nested rectangles are drawn.

Example

```
10  SCREEN 3:COLOR 1,4,4:CLS
20  FOR X=1 TO 8
30  FOR Y=1 TO 8
40  C%=-15*( (X+Y)MOD2=0)-( (X+Y) MOD2=1)
50  LINE (16*X,16*Y) - (16*X+15,16*Y+15),C%,BF
60 NEXT Y,X
70 GOTO 70
```

Line 10: selects graphics mode 3, foreground black, background and border dark blue and clears the screen.
Line 20: sets up a loop to be executed for values of X from 1 to 8.
Line 30: sets up another loop, to be executed for values of Y from 1 to 8.
Line 40: sets C% to 15 if (X+Y) is even, 1 if (X+Y) is odd.
Line 50: draws a solid rectangle in colour C%, top left-hand corner (16*X,16*Y), bottom right-hand corner (16*X+15,16*Y+15).
Line 60: end of Y loop, end of X loop.
Line 70: holds the display on the screen.

This program draws a chessboard. It will work equally well in graphics mode 2.

Exercise 10.1

Write a program which will produce a random kaleidoscope pattern of coloured dots, symmetrical about the lines X=128 and Y=96, in mode 3.

Exercise 10.2

(a) Write a subroutine, using only relative co-ordinates, which will

draw a solid square with sides twenty pixels long, and a cross (+) made up of two lines each twenty pixels long on top of this square.

(b) Using this subroutine, write a program to draw four squares with crosses on top, the top left-hand corners of the squares being the points (50,50),(150,50),(50,100) and (150,100), in magenta on a white background.

10.5 CIRCLES, ARCS AND ELLIPSES

The **CIRCLE** command is used to draw circles, arcs and ellipses. You must specify the centre of the circle, in absolute or relative co-ordinates, and the radius (in pixels). There are several other optional parameters; if you omit any of these, remember to put in the commas following them if later parameters are to be specified.

The syntax for the command is:

CIRCLE(<co-ordinates of centre>),<radius>,[<colour>],[<start angle>],[<end angle>][,<aspect ratio>]

The X co-ordinate, Y co-ordinate and radius must all be within the integer range. If they are chosen so that not all of the circle will fit on the screen, only the part which will fit is drawn.

Example

```
10 SCREEN 2: COLOR 15,1,1:CLS
20 FOR X=-40 TO 280 STEP 10
30 CIRCLE(X,50),30
40 NEXT X
50 GOTO 50
```

Line 10: selects graphics mode 2, foreground white, background and border black, and clears the screen.
Line 20: sets up a loop to be executed for values of X from –40 in increments of 10 to 280.
Line 30: draws a white circle, centre (X,30), radius 30 pixels.
Line 40: end of loop.
Line 50: holds the display on the screen.

A horizontal row of overlapping circles is drawn across the screen,

running off either side.

The colour parameter, if specified, will produce a circle in the appropriate colour.

The start and end angle parameters are used for drawing arcs. The angles are measured in radians, in an anti-clockwise sense starting from the X axis (see Figure 5). As radians are required, it is useful to put in a command specifying the value of PI (π) before drawing an arc:

```
PI=4*ATN(1)
```

The angles must both be in the range 0 to 2*PI (2 π). The end angle need not be greater than the start angle: to draw the right-hand half of a circle, use a start angle of 1.5*PI (3 π/2) and an end angle of 0.5*PI (π/2).

If either or both of the angles is preceded by a minus sign, lines will be drawn joining the centre of the circle to the appropriate end(s) of the arc. However, the computer cannot distinguish between –0 and 0; if you want to join the centre of the circle to the start of an arc whose start angle is 0, replace the 0 with –2*PI.

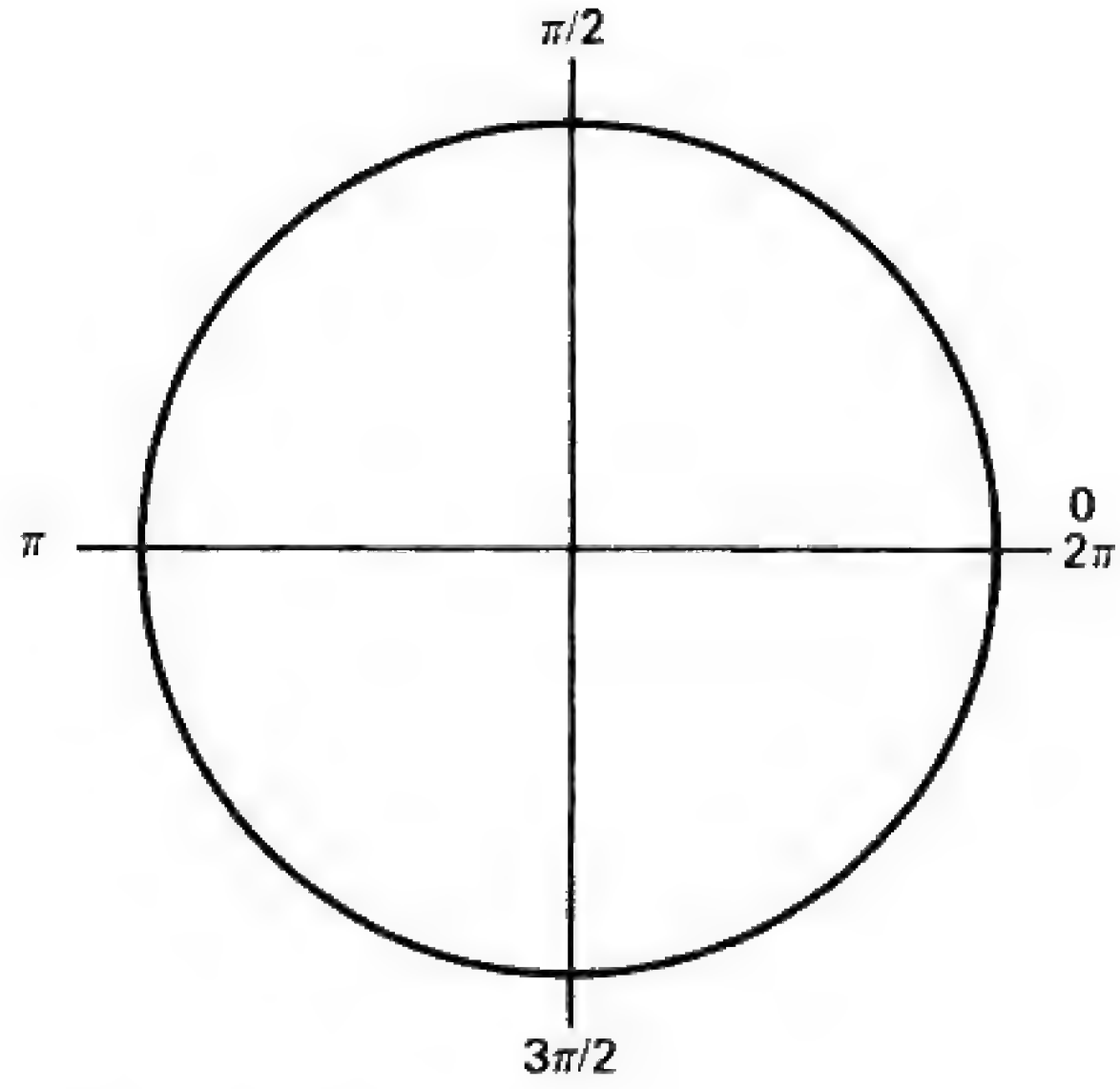


Figure 5 *Circle start and end angles*

Example

```
10 SCREEN 2:COLOR 1,14, 14:CLS
20 PI=4*ATN(1)
30 CIRCLE (50,50),50,,1.5*PI,2*PI
40 CIRCLE (150,50),50,,PI,1.5*PI
50 CIRCLE (150,150),50,,.5*PI,PI
60 CIRCLE (50,150),50,,0,.5*PI
70 GOTO 70
```

Line 10: selects graphics mode 2, foreground black, background and border grey, and clears the screen.
Line 20: defines the value of PI.
Line 30: draws the bottom right quadrant of a circle centre (50,50), radius 50, in black.
Line 40: draws the bottom left quadrant of a circle centre (150,50), radius 50, in black.
Line 50: draws the top left quadrant of a circle centre (150,150), radius 50, in black.
Line 60: draws the top right quadrant of a circle centre (50,150), radius 50, in black.
Line 70: holds the display on the screen.

This will produce a diamond shape with its sides curving inwards.

Example

```
10 SCREEN 2:COLOR 1,14,14:CLS
20 PI=4*ATN(1)
30 CIRCLE(100,100) ,50,5,-.3*PI,-.7*PI
40 GOTO 40
```

Line 10: selects graphics mode 2, foreground black, background and border grey, and clears the screen.
Line 20: defines the value of PI.
Line 30: draws an arc of a circle centre (100,100), radius 50, in light blue and draws lines joining both ends of the arc to the centre of the circle.
Line 40: holds the display on the screen.

This will draw a pie with a slice cut out of it.

The aspect ratio determines the ratio of the vertical radius to the horizontal radius. The default value is 1, when a circle is drawn. If the aspect ratio is greater than 1, a vertically elongated ellipse is drawn; the

specified radius is taken to be the vertical radius. If the aspect ratio is less than 1, a horizontally elongated ellipse is drawn; the specified radius is taken to be the horizontal radius.

On some televisions, circles drawn with the CIRCLE command and the default aspect ratio of 1 appear elliptical. In this case, the aspect ratio must be altered (generally increased) to produce true circles.

Example

```
10 REM FACE
20 SCREEN 2:COLOR9,15,15:CLS
30 PI=4*ATN(1)
40 CIRCLE(120,90),80,,,,1.8: REM OUTLINE
50 CIRCLE(100,80),10,7,,,0.4:CIRCLE(140,80),10,7,,,0.4:REM
   EYES
60 CIRCLE(100,80),4,1:CIRCLE(140,80),4,1:REM PUPILS
70 CIRCLE(120,120),20,8,PI,2*PI:REM MOUTH
80 LINE(120,80)-(126,110):LINE-(110,110):REM NOSE
90 GOTO 90
```

Line 10: comment.
Line 20: selects graphics mode 2, foreground light red, background and border white, and clears the screen.
Line 30: defines the value of PI.
Line 40: draws a light red ellipse centre (120,90), vertical radius 80, horizontal radius 80/1.8.
Line 50: draws two sky blue ellipses, centres (100,80) and (140,80), horizontal radii 7, aspect ratio 0.4.
Line 60: draws two black circles, centres (100,80) and (140,80), radii 4.
Line 70: draws the bottom half of a circle centre (120,120), horizontal radius 20, in medium red.
Line 80: draws lines from (120,80) to (126,110) and from (126,110) to (110,110) in light red.
Line 90: holds the display on the screen.

This program draws a simplified face. You could extend it to add other features such as hair and eyebrows.

10.6 PAINTING

A solid rectangle can be produced by adding the letter F to the end of

the appropriate line command, but there is no similar extension to the CIRCLE command to enable a solid disc to be drawn. To fill a circle, or any other closed shape, the **PAINT** command must be used. This takes the form:

PAINT(<co-ordinates of start point>)[,<paint colour>][,<border colour>]

Absolute or relative co-ordinates can be used. If the paint colour is omitted, the current foreground colour will be used. In mode 2, the border must be the same colour as the paint; in mode 3, a different border colour may be specified. The painting starts at the point whose co-ordinates are given, and stops when the border is reached, or if there is no border, or a gap in the border, continues over the whole screen.

Example

```
10 SCREEN 2:COLOR 15,14:CLS
20 LINE(20,20)-(200,160),,BF
30 CIRCLE(110,90),30,6
40 PAINT(110,90),6
50 GOTO 50
```

Line 10: selects graphics mode 2, foreground white, background and border grey, and clears the screen.
Line 20: draws a solid white rectangle, top left corner (20,20), bottom right corner (200,160).
Line 30: draws a dark red circle centre (110,90), radius 30.
Line 40: paints the circle dark red.
Line 50: holds the display on the screen.

A white rectangle with a red disc on it – the Japanese national flag.

Example

```
10 SCREEN 2:COLOR 1,15,15:CLS
20 PI=4*ATN(1)
30 CIRCLE(120,90),80,8,-2*PI,-0.8*PI:PAINT(120,80),8
40 CIRCLE(120,90),80,7,-0.8*PI,-1.4*PI:PAINT(110,90),7
50 CIRCLE(120,90),80,2,-1.4*PI,-2*PI:PAINT(130,100),2
60 GOTO 60
```

Line 10: selects graphics mode 2, foreground black, background and border white, and clears the screen.

Line 20: defines the value of PI.
Line 30; draws in red an arc of a circle centre (120,90), radius 80, with both ends of the arc joined to the centre of the circle, and paints this shape red.
Line 40: draws in sky blue another arc of the same circle, with both ends joined to the centre, and paints this shape sky blue.
Line 50: draws in green the remainder of the circle, again with both ends joined to the centre, and paints this shape green.
Line 60: holds the display on the screen.

This program should produce a pie chart with red, blue and green sections. But if you run it, you will see that there is a problem: the borders between the sections are smudgy. When the blue section is drawn, the red/blue border area already contains two colours, red and white, so the introduction of a third colour, blue, causes smudging. Similarly, the blue/green border area already contains blue and white before the green is added. The smudging can be eliminated by painting the whole circle red, then painting both the blue section and the green section blue, then adding the green on top of the blue. Change lines 30 and 40 to:

```
30 CIRCLE(120,90),80,8:PAINT(120,90),8
40 CIRCLE(120,90),80,7,-0.8*PI:PAINT(120,100),7
```

Now the pie chart will be drawn clearly.

Example

```
10 SCREEN 3:COLOR 15,1,1:CLS
20 FOR R=97 TO 9 STEP -8
30 CIRCLE(126,98),R
40 PAINT(126,98),R/8,15
50 NEXT R
60 GOTO 60
```

Line 10: selects graphics mode 3, foreground white, background and border black, and clears the screen.
Line 20: sets up a loop to be executed for values of R from 97 in decrements of 8 to 9.
Line 30: draws a white circle centre (126,98), radius R.
Line 40: paints the circle in colour R/8 (border colour white).
Line 50: end of loop.
Line 60: holds the display on the screen.

This program produces a pretty pattern of alternate white and coloured concentric rings.

Exercise 10.3

Write a program to draw a blue circle inside a red triangle inside a green square, on a white background. (All the shapes should be solid.)

Exercise 10.4

```
10 SCREEN 2:CLS
20 FOR A=1 TO 5
30 CIRCLE(128,96),50,,,,A
50 NEXT A
60 GOTO 60
```

This program will draw a pattern of concentric ellipses. Add another line (line 40) to the program, to superimpose on this the same pattern rotated through 90° (PI/2(π/2) radians).

10.7 DRAW STRINGS

The *graphics macro language* (GML) offers a large selection of single-character commands, which are utilized by putting them in a string after the BASIC keyword **DRAW**. They enable you to produce complicated pictures with the minimum of programming.

The whole DRAW string must be enclosed in double quotation marks. The commands within the string may be separated by semicolons, but these are not essential, except after variable names. The string can be up to 255 characters long.

The parameters for the individual commands may be numeric constants or numeric variables. Variables must be distingished by preceding them with an equals sign, and following them with a semicolon.

The following commands are used to specify the directions in which lines are to be drawn:

U	up
D	down
L	left
R	right
E	diagonally up and right
F	diagonally down and right

G diagonally down and left
H diagonally up and left

Each of these commands must be followed by a numeric constant or variable. With U, D, L and R, the number gives the length of the line to be drawn in pixels. With E, F, G and H, it gives the length of the distance moved along the X and Y axes. Thus U5 means 'draw a line five pixels long to the right'; E5 means 'draw a diagonal line to a point five pixels up and five pixels right of the present graphics cursor position.'

Example

```
10 SCREEN 2:COLOR 15,6,6:CLS
20 PSET (20,20)
30 DRAW"R20D20L20U20"
40 PSET(100,20)
50 DRAW"F20G20H20E20"
60 GOTO 60
```

Line 10: selects graphics mode 2, foreground white, background and border dark red, and clears the screen.
Line 20: moves the graphics cursor to the point (20,20) (and sets this point to white).
Line 30: draws a line 20 pixels right, 20 pixels down, 20 pixels left and 20 pixels up.
Line 40: moves the graphics cursor to the point (100,20) (and sets this point to white).
Line 50: draws a line diagonally right and down 20, diagonally left and down 20, diagonally left and up 20, diagonally right and up 20.
Line 60: holds the display on the screen.

The diamond drawn by line 50 is noticeably larger than the square drawn by line 30.

Example

```
10 SCREEN 2:COLOR 14,13,13:CLS
20 PSET (10,10)
30 FOR A%=10 TO 60 STEP 10
40 DRAW"F=A%;U10"
50 NEXT A%
60 GOTO 60
```

Line 10: selects graphics mode 2, foreground grey, background and border magenta, and clears the screen.
Line 20: moves the graphics cursor to the point (10,10) (and sets this point to grey).
Line 30: sets up a loop to be executed for values of A% from 10 in increments of 10 to 60.
Line 40: draws a line from the current graphics cursor position diagonally right and down A%, then up 10.
Line 50: end of loop.
Line 60: holds the display on the screen.

The command **M** can be used to draw lines at intermediate angles. M is followed by X and Y co-ordinates; these are not enclosed in brackets, but must be separated by a comma. You can use absolute co-ordinates, or relative co-ordinates, specified by a + or – prefix:

M50,100 means draw a line to the point (50,100).
M+10,–20 means draw a line to a point 10 pixels right and 20 pixels up from the present cursor position.

All these commands normally move the graphics cursor to the end of the line which is drawn. They can be prefixed by the letter **B** (blank), when the cursor is moved but no line is drawn. Most DRAW strings start with a BM command, to move the cursor to the required start position. This is more convenient than using PSET to move the cursor.

Example

```
10 SCREEN 2:COLOR 1,15,15:CLS
20 DRAW"BM100,20;R50M+22,+44M-22,+44L50M-22,-
   44M+22,-44"
30 GOTO 30
```

Line 10: selects graphics mode 2, foreground black, background and border white, and clears the screen.
Line 20: moves the graphics cursor to the point (100,20), then draws lines 50 pixels right, 22 pixels right and 44 pixels down, 22 pixels left and 44 pixels down, 50 pixels left, 22 pixels left and 44 pixels up, and 22 pixels right and 44 pixels up, all in black.
Line 30: holds the display on the screen.

A black hexagon is drawn on a white background.

The prefix N before one of the above commands returns the graphics cursor to the start of the line.

Example

```
10 SCREEN 2:COLOR 11,4,4:CLS
20 DRAW"BM100,50R50NG50D50NH50L50U50"
30 GOTO 30
```

Line 10: selects graphics mode 2, foreground light yellow, background and border dark blue, and clears the screen.
Line 20: draws a square with sides 50 pixels long, top left-hand corner (100,50), and both its diagonals in yellow.
Line 30: holds the display on the screen.

The examples given so far have used the COLOR command to set the screen foreground and background colours; the drawing has been done in the foreground colour. To change the drawing colour, use the command **C** followed by a colour code. The colour set by the C command remains effective for all DRAW commands until changed by another C command, or by the use of another graphics command. The foreground colour set by the last COLOR command remains the default colour for other graphics commands.

Example

```
10 SCREEN 2:COLOR 11,4,4:CLS
20 DRAW"C2BMO,120R255":PAINT(0,191),2
30 DRAW"C6BM88,180U100E50F50D100L100":PAINT(100,160),6
40 DRAW"BM40,180;U80R8D80L8":PAINT(44,170),6
50 CIRCLE(44,60),40,12:PAINT(44,60),12
60 CIRCLE(200,30),20:PAINT(200,30)
70 DRAW"BM136,180;U40R20D40L20":PAINT(140,170),11
80 GOTO 80
```

Line 10: selects graphics mode 2, foreground light yellow, background and border dark blue, and clears the screen.
Line 20: draws a horizontal line across the screen from the point (0,120) in medium green, and paints the area below this line green.
Line 30: sets the drawing colour to dark red, draws a house shape and paints it red.
Line 40: draws a treetrunk and paints it red.
Line 50: draws a dark green circle and paints it dark green (this is the top of the tree).

Line 60: draws a yellow circle and paints it yellow (the sun).
Line 70: draws the door of the house in yellow, and paints it yellow
Line 80: holds the display on the screen.

The scale of the drawing produced by a DRAW string can be set by the command S. This is followed by a number between 0 and 255. The default value is 4, so S4 produces a normal scale drawing. S8 produces a drawing twice the normal size, S12 produces a drawing three times the normal size, and so on. S0 is the same as S4.

Example

```
10  10 SCREEN 2:COLOR 1,5,5
20  FOR S%=1 TO 12
30  CLS
40  Y%=50+8*S%
50  DRAW"BM120,=Y%;S=S%;L8D4L4U4L4U16R4U20R24D20
    R4D16L4D4L4U4L8"
60  FOR D=0 TO 100:NEXT D
70  NEXT S%
80  GOTO 80
```

Line 10: selects graphics mode 2, foreground black, background and border light blue.
Line 20: sets up a loop to be executed for values of S% from 1 to 12.
Line 30: clears the screen.
Line 40: defines the start Y co-ordinate for the drawing.
Line 50: draws a rough outline of a car, to scale S%.
Line 60: delay loop.
Line 70: end of loop.
Line 80: holds the last drawing on the screen.

The increasing scale of the drawings creates the impression that the car is coming towards you.

The scale factor affects the length of lines drawn using the U, D, L, R, E, F, G and H commands, and of lines drawn using the M command where relative co-ordinates are specified. It does not have any effect on lines drawn using the M command with absolute co-ordinates. You should, therefore, only use absolute co-ordinates to specify the start point if your DRAW string is to include a scale factor.

The scale factor remains effective until it is cancelled by another scale command. It is not even reset at the end of a program! It is, therefore, advisable to set the scale at the start of every program which uses

DRAW strings, even if only the default value is used within the program.

Try to ensure that the lengths of all lines will be integers after scaling. If the scale factor is a multiple of 4, no problems should arise; if the scale factor is not a multiple of 4, then all specified lengths in the DRAW string should be multiples of 4. This is particularly important if the drawing is to be PAINTed. If the lengths of lines have to be rounded, gaps may appear in the borders of previously closed shapes, and paint can spill through these gaps.

Example

```
10 SCREEN 2:COLOR 1,7,7:CLS
20 DRAW"S4BM20,20R40D40L10U10L10U10L10U10L10U10"
30 PAINT(21,21)
40 GOTO 40
```

Line 10: selects graphics mode 2, foreground black, background and border sky blue, and clears the screen.
Line 20: draws a closed outline.
Line 30: paints the outline black.
Line 40: holds the display on the screen.

Try changing the scale factor in line 20. Even numbers produce the same painted shape in various sizes; with odd numbers, the outline is not closed and the whole screen is painted black.

The angle command, **A**, can be used to rotate the drawing:

A0	draws in the normal position.
A1	rotates anti-clockwise through 0.5*PI ($\pi/2$) radians (90°).
A2	rotates through PI (π) radians (180°).
A3	rotates anti-clockwise through 1.5*PI ($3\pi/2$) radians (270°).

Only the above options are available; this command cannot be used to rotate a drawing through an angle of less than 90°.

Example

```
10 SCREEN 2:COLOR 1,7,7:CLS
20 CIRCLE(120,90),90
30 FOR A%=3 TO 0 STEP -1
40 DRAW"S4A=A%;C1;U80NG10NF10"
50 FOR D=0 TO 200:NEXT
60 DRAW"C7;NF10NG10D80"
```

```
70 NEXT A%
80 GOTO 30
```

Line 10: selects graphics mode 2, foreground black, background and border sky blue, and clears the screen.
Line 20: draws a black circle centre (120,90), radius 90.
Line 30: sets up a loop to be executed for values of A% from 3 in decrements of 1 to 0.
Line 40: draws a black arrow shape at angle A%.
Line 50: delay loop.
Line 60: draws over the arrow shape in the background colour.
Line 70: end of loop.
Line 80: returns to the start of the loop.

An arrow-shaped pointer moves clockwise around a circular dial. The speed of rotation can be set by adjusting the delay loop (line 50).

Strings of commands in the graphics macro language can be stored and manipulated in the same way as other character strings. If the same string of commands is to be used more than once, it is useful to assign it to a string variable:

```
10 SCREEN 2:CLS
20 A$="R10D10L10U10"
30 DRAW"BM10,10"+A$
40 DRAW"BM30,30"+A$
50 GOTO 50
```

Line 10: selects graphics mode 2, clears the screen.
Line 20: assigns to the variable A$ the commands to draw a small square.
Line 30: draws the square, starting at (10,10).
Line 40: draws the square, starting at (30,30).
Line 50: holds the display on the screen.

Predefined substrings can also be executed within a DRAW string by using the X command. This is followed by the name of the variable to which the substring has been assigned, then a semicolon.

Example

```
10  SCREEN 2:COLOR 15,2,2:CLS
20  A$="R4L2M+4,-8NR8M+4,-8M+8,+16L2R4":D$=
    "E8F8G8H8"
```

```
30 LINE(50,20)-(148,170),,BF
40 DRAW"S4A0C8BM56,40XA$;BM56,60XD$;S8BM84,95XD$;"
50 DRAW"S4BM126,130XD$;A2BM142,150XA$;"
60 PAINT(60,60),8,8:PAINT(90,95),8,8:PAINT(130,130),8,8
70 GOTO 70
```

Line 10: selects graphics mode 2, foreground white, background and border green, and clears the screen.
Line 20: assigns to A$ a string of commands which will produce a letter A, and to D$ a string of commands to produce a diamond shape.
Line 30: draws a solid white rectangle.
Line 40: draws in red the letter A in the top left-hand corner of the white rectangle, a small diamond beneath it and a larger diamond in the centre of the rectangle.
Line 50: draws a small diamond near the bottom right-hand corner of the rectangle and an upside-down letter A beneath it.
Line 60: paints the diamonds red.
Line 70: holds the display on the screen.

This program produces a picture of a playing card – the ace of diamonds.

Exercise 10.5

Write a program to draw the word 'HELLO' along each side of the screen as shown in Figure 6, using a substring for the letter L.

Exercise 10.6

Design a spaceship, and write a program to draw it.

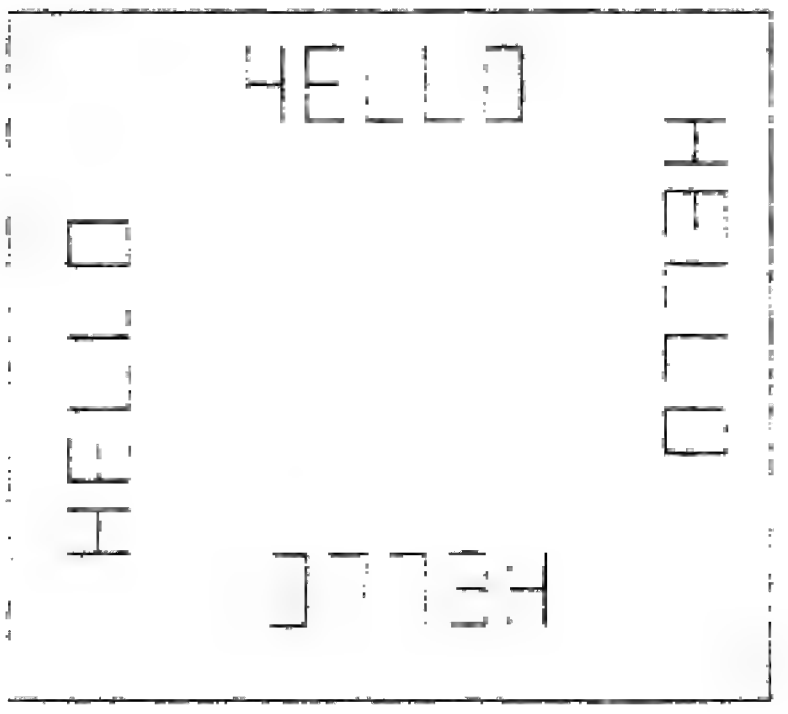


Figure 6

11

Function Keys, Cursor Keys, Joysticks and Interrupts

11.1 THE FUNCTION KEYS

The contents of the function keys are normally displayed at the bottom of the text screen. This display can be switched off by the command KEY OFF and on again by the command KEY ON. The key contents can also be listed on the screen by using the command **KEY LIST**. The initial contents of the ten function keys (five keys and five SHIFTed keys) are:

function keys	*SHIFTED function keys*
F1 color<space>	F6 color 15,4,4<RETURN>
F2 auto<space>	F7 cload"
F3 goto<space>	F8 cont<RETURN>
F4 list<s>	F9 list. <RETURN><CRSR UP><CRSR UP>
F5 run<RETURN>	F10 <CLS>run<RETURN>

These contents can be changed by using the command **KEY**, which takes the form:

KEY <function key no.>,<string>

The contents of the string are assigned to the specified function key. The string can contain any of the MSX characters, including control characters (see Appendix E for a list of control codes). If the key definition ends with CHR$(13) the command assigned to the key will auto-execute, i.e. there will be no need to press <RETURN> after the function key.

Example

```
KEY 1,"RENUM"+CHR$(13)
```

Enter this in direct mode and you will see the word 'color' at the bottom of the screen change to 'RENUM'. Now when you press F1, your program will be renumbered.

If you assign the keywords you use most often to the function keys, they can save you a good deal of time when typing in programs.

11.2 CURSOR KEYS AND JOYSTICKS

All MSX computers have four cursor keys, marked with arrows up, down, left and right. These are normally, but not invariably, arranged in a diamond shape on the right of the keyboard. There are also one or two joystick ports, into which any MSX compatible joysticks can be plugged. The cursor keys and joysticks are used to move objects around the screen, particularly in arcade games. A special function, **STICK**, can be used to determine which direction has been selected.

The argument of the function can be 0, 1 or 2. STICK(0) returns the direction of the cursor keys, STICK(1) returns the direction of joystick 1 and STICK(2) returns the direction of joystick 2. If the joystick is in the neutral position (or no cursor keys are being pressed), the function returns the value 0. Otherwise, its value is as shown:

1 up
2 diagonally up and right
3 right
4 diagonally right and down
5 down
6 diagonally down and left
7 left
8 diagonally left and up

Note that to move diagonally, you must press two cursor keys at once.

Once the direction has been read, an ON...GOTO or ON...GOSUB command can be used to transfer execution to the appropriate movement routine.

Example

```
10 CLS
20 PRINT"(J)OYSTICK OR (C)URSOR
   KEYS?":A$=INPUT$(1)
```

```
30 IF A$="J" OR A$="j" THEN D=1:GOTO 50
40 IF A$="C" OR A$="c" THEN D=0 ELSE GOTO 20
50 SCREEN 2:COLOR 1,15,15:CLS
60 ON STICK(D) GOSUB 80,90,100,110,120,130,140,150
70 GOTO 60
80 DRAW"U1":RETURN
90 DRAW"E1":RETURN
100 DRAW"R1":RETURN
110 DRAW"F1":RETURN
120 DRAW"D1":RETURN
130 DRAW"G1":RETURN
140 DRAW"L1":RETURN
150 DRAW"H1":RETURN
```

Line 10: clears the text screen.
Line 20: prints prompt, waits for input of a single character, and assigns this character to A$
Line 30: if the J key (normal or SHIFTed) was pressed, sets D to 1 and jumps to line 50.
Line 40: if the C key (normal or SHIFTed) was pressed, sets D to 0; otherwise, returns execution to line 20.
Line 50: selects graphics mode 2, foreground black, background white, and clears the screen.
Line 60: checks value returned by cursor keys or joystick 1, and jumps to appropriate subroutine.
Line 70: returns to check cursor keys or joystick again.
Lines 80–150: subroutines draw a line in the appropriate direction, then return to line 70.

This program allows you to doodle on the screen, using the cursor keys or a joystick in port 1. (A more sophisticated version is given in Chapter 19.)

Each of the two joysticks has two fire buttons. The **STRIG** function returns the status of the space bar (for use with the cursor keys) and fire buttons, giving a value of –1 if they are pressed and 0 otherwise. The argument of the function is:

0	space bar
1	joystick 1 fire button 1
2	joystick 2 fire button 1
3	joystick 1 fire button 2
4	joystick 2 fire button 2

Example

```
10 PRINT "PRESS THE SPACE BAR"
20 IF STRIG(0)=-1 THEN BEEP
30 GOTO 20
```

Line 10: prints instructions.
Line 20: beeps if space bar is depressed.
Line 30: returns to line 20.

This program will produce a beep every time the space bar is pressed.

11.3 INTERRUPTS

MSX BASIC includes a wide range of *interrupt-handling* commands. These can be used to instruct the computer to look out for certain events happening, and to interrupt the normal running of the program and jump to a special subroutine if one of these events occurs. Interrupts can be triggered by any of the following events:

1. Set time intervals.
2. Function key press.
3. <CTRL> and <STOP> press.
4. Space bar or joystick fire button press.
5. Sprite collision.

To use interrupts you must put two different commands in your program, one to *specify* the interrupt routine, i.e. to tell the computer which subroutine to go to when the interrupt occurs, and another to *enable* the interrupt, i.e. to tell the computer to start watching for the interrupt triggering event to occur. You can also *disable* the interrupt, so that, for example, the interrupt is only functional during a particular section of the program, or hold the interrupt, so that the computer knows that the event has occurred but does not execute the interrupt subroutine until it is enabled again.

The commands used to specify interrupt routines take the form:

ON <event which triggers interrupt> **GOSUB** <line no.>

The interrupt routines have the same structure as ordinary subroutines, i.e. they must end with RETURN.

To enable an interrupt, use:

<event> **ON**

To disable the interrupt:

<event> **OFF**

To hold the interrupt:

<event> **STOP**

The subroutine must be specified before the interrupt is enabled, or the computer will not know where to go to when the event occurs!

An automatic <event> STOP occurs at the start of an interrupt subroutine, so that execution of the subroutine is not itself interrupted. This STOP is disabled again at the end of the subroutine, and as it only halts execution of the interrupt rather than preventing it, the subroutine will be executed again immediately if the triggering event has occurred again. To prevent the subroutine from being executed several times in succession, you can disable the interrupt at the start of the subroutine; it will not then be automatically re-enabled at the end, so you must also include a command to re-enable it.

Interrupts can be used only in a program, not in direct mode. They are also disabled during the execution of error-trapping routines (see Section 15.1).

Set Time Interval Interrupts

The computer is fitted with an internal clock, which is set to 0 when the computer is switched on and is incremented 50 times every second. The TIME function can be used to read the value of the timer (see Section 7.3), and also to set its value:

```
TIME=0
```

resets the timer to 0.

Interrupts can be set to occur at fixed time intervals: an interval of 50 will cause interrupts to occur every second, an interval of 200 will produce an interrupt every four seconds, and so on. The command used to specify the interrupt subroutine is:

ON INTERVAL=<length of interval> **GOSUB** <line no.>

and the commands used to enable/disable/halt the interrupt are:

INTERVAL ON/OFF/STOP

Set time interval interrupts can be used to display a digital clock in the corner of the screen while a program is running. The clock could display the elapsed time or, if you include an appropriate input routine, the actual time. This routine will display the elapsed time since the program started to run:

```
 10 ON INTERVAL=50 GOSUB 1000
 20 S%=0:M%=0
 30 INTERVAL ON
 40 REM MAIN PROGRAM STARTS HERE
    ”
    ”
    ”
980 GOTO 980
990 REM INTERRUPT ROUTINE
1000 S%=S%+1:IF S%=60 THEN S%=0:M%=M%+1
1010 LOCATE 30,0
1020 PRINT M%;“:”;:PRINT USING “##”;S%
1030 RETURN
```

Line 10: specifies interval interrupt routine (starting at line 1000), to be executed at one-second intervals.
Line 20: sets seconds (S%) to 0, minutes (M%) to 0.
Line 30: enables interval interrupt
Line 980: stops execution passing accidentally to the interrupt routine.
Line 990: comment.
Line 1000: (start of interrupt routine): increments S% (number of seconds). If S% has reached 60, increments M% (number of minutes) and resets S% to 0.
Line 1010: locates text cursor at column 30, row 0.
Line 1020: prints number of minutes, then a colon, then number of seconds (using two figures).
Line 1030: returns execution to main program routine.

Function Key Interrupts

Interrupts can be triggered by any or all of the function keys. The

command used to specify the subroutines for the function keys is:

ON KEY GOSUB <line no.>,<line no.>,...

The first line number given specifies the subroutine for key F1, the second the subroutine for key F2, and so on. Any of the numbers can be omitted, but if you want to specify, say, a subroutine for key F3 only, you must precede the line number by two commas to indicate that subroutines for F1 and F2 have not been specified.

A separate command must be used to enable, disable or halt interrupts by each function key. These commands take the form:

KEY (<function key no.>) **ON/OFF/STOP**

If you want to enable or disable interrupts by several function keys at once, use a FOR...NEXT loop:

```
FOR C=1 TO 5:KEY(C) ON:NEXT C
```

will enable keys F1 to F5.

<CTRL> and <STOP> Interrupts

Pressing <CTRL> and <STOP> together normally stops the execution of a program and puts the computer into command mode. If you want to prevent a program from being stopped in this way, you can set up an interrupt routine which is triggered by <CTRL> and <STOP>. This may be an 'empty' routine, which contains no commands other than the essential RETURN, or a routine which prints out a message warning users that the program cannot be broken into.

The commands to use are:

ON STOP GOSUB <line no.>
STOP ON/OFF/STOP

Bear in mind that using a break-proofing routine will make it impossible for you to stop the program running to edit the listing, or to save it on cassette. You should, therefore, leave out the command which enables the interrupt until you have debugged the program; after inserting STOP ON, SAVE the program before you RUN it.

Example

```
10  ON STOP GOSUB 1000
20  STOP ON
30  REM Main program starts here
    ''
    ''
    ''
 990  REM Interrupt routine
1000  RETURN
```

Line 10: specifies <CTRL> and <STOP> interrupt routine.
Line 20: enables <CTRL> and <STOP> interrupt.
Line 1000: returns execution to main program routine (pressing <CTRL> and <STOP> has no effect).

Space Bar and Joystick Fire Button Interrupts

The **ON STRIG GOSUB** and **STRIG(<no.>) ON/OFF/STOP** commands are used to set up space bar and fire button interrupts. They are used in much the same way as the function key interrupt commands: ON STRIG GOSUB is followed by up to five line numbers, specifying the routines for space bar, joystick 1 fire button 1, joystick 2 fire button 2, joystick 2 fire button 1, and joystick 2 fire button 2 interrupts, respectively. The number specified in the STRIG(n) ON/OFF/STOP is the same as the argument of the corresponding STRIG function: STRIG(0) ON enables space bar interrupts, STRIG(1) ON enables interrupts by joystick 1 fire button 1, etc.

Example

```
10  ON STRIG GOSUB 40,50,60,70,80
20  FOR C=0 TO 4:STRIG(C) ON:NEXT C
30  GOTO 30
40  CLS:PRINT"FIRE 0":RETURN
50  CLS:PRINT"FIRE 1":RETURN
60  CLS:PRINT"FIRE 2":RETURN
70  CLS:PRINT"FIRE 3":RETURN
80  CLS:PRINT"FIRE 4":RETURN
```

Line 10: specifies space bar and fire button interrupt routines.
Line 20: enables all the space bar and fire button interrupts.
Line 30: waits for interrupt to be triggered.

Lines 40–80 (interrupt routines): print out appropriate message, return to line 30.

Use this program to find out which of the fire buttons on your joystick is which.

Sprite Collision Interrupts

Sprites are discussed in Chapter 14. Interrupts can be triggered by two or more sprites overlapping, using the **ON SPRITE GOSUB** and **SPRITE ON/OFF/STOP** commands.

Exercise 11.1
Give the statements which would be used to assign the keywords GOSUB and RETURN to <F1> and <F2> respectively.

Exercise 11.2
Write a program protection routine which will print this message when you press <CTRL> and <STOP>:

```
THIS PROGRAM IS BREAK-PROOFED
PRESS ANY KEY TO CONTINUE
```

and will stop the program only if you then press S.

12 Sound

12.1 KEY CLICK

When you type on the MSX keyboard, a clicking sound is produced as each key is depressed. If you hold a key down so that it auto-repeats, there is another click as each symbol appears on the screen. Some people find that this click makes it easier to type accurately, but others find it irritating. The key click can be switched on and off by an extension to the **SCREEN** command. SCREEN can be followed by up to five different parameters. The first parameter, as we have seen, selects the screen mode: text mode 0 or 1, or graphics mode 2 or 3. The functions of the second, fourth and fifth parameters will be explained later. The third parameter is the key click switch. The click is disabled if this parameter is 0, and enabled if it is any other number:

```
SCREEN 0,,0 gives text mode 0 with no key click.
SCREEN 0,,1 gives text mode 0 with key click.
```

Try these out in direct mode to see which you prefer.

12.2 BEEP

The simplest BASIC command which produces a sound is **BEEP**, which does just what its name suggests. It produces a beep lasting 0.04 seconds. The sound is fed through the television speaker, so its volume will depend on the setting of the volume control on the television. The same beep can be produced by control code 7: PRINT CHR$(7).

Beeps are very useful for attracting attention. You could put them in

your programs to indicate when a keypress is required, or to let you know when a lengthy series of calculations has been completed.

12.3 PLAYING MUSIC

MSX computers have three sound channels, which can be used to play music in three-part harmony. Music can be programmed using the *music macro language* (MML), which is very similar to the graphics macro language (see Chapter 10). The commands are put in a string following the keyword **PLAY**.

The three channels each have a range of eight octaves. The first (lowest) note in each octave is C, followed by C# (or D♭), D, D# (or E♭), E, F, F# (or G♭), G, G# (or A♭), A, A# (or B♭) and finally B. These notes are shown on one octave of a piano keyboard in Figure 7.

The commands **A**, **B**, **C**, **D**, **E**, **F** and **G** play the corresponding notes, with + (or #) or – signs after the letter being used to indicate sharps and flats respectively:

PLAY"CDEFGAB" will play all but the last note of the scale of C major.

You cannot use C+, D–, E+ or F– as these do not exist as separate notes.

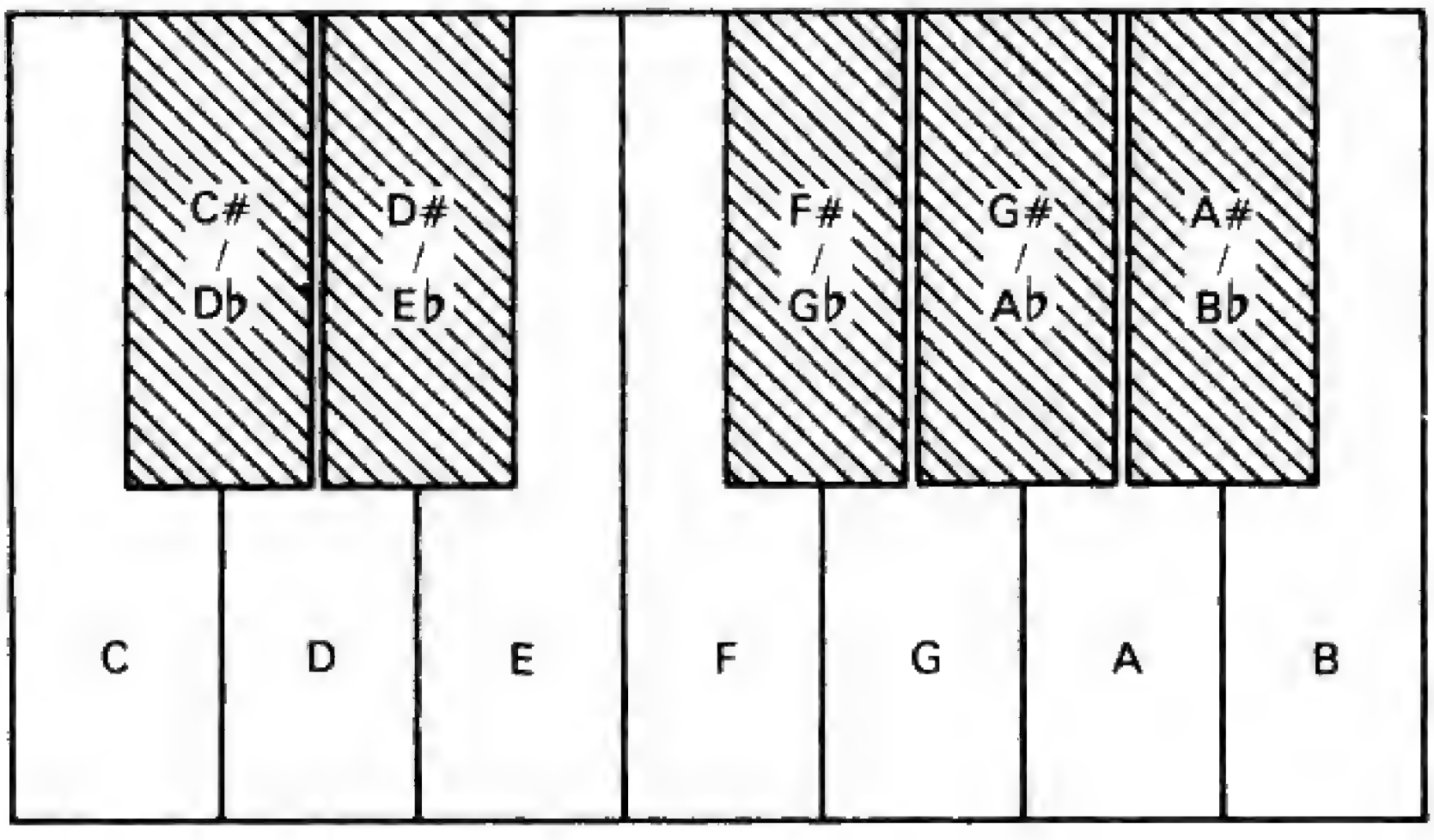


Figure 7 *A piano keyboard*

The octave can be set by the command **O** followed by a number from 1 to 8; the default is O4. Be very careful to distinguish between the letter O and the number 0, which appear almost identical in some program listings. Each octave command remains effective until another octave command is used:

PLAY"O4EF+G+ABO5C+D+E" will play an octave of the scale of E major.

Remember that the octave change always comes between the notes B and C, whatever key the music is in.

Numeric variables can be used instead of numbers in all the MML commands which require numeric arguments. The variable name must be preceded by an = sign, and followed by a semicolon.

Notes can also be played using the **N** command, followed by a number from 1 to 96. N1 is C+ in octave 1, N2 is D in octave 1, and so on. C in octave 1 is not available with this command, as N0 is used as a rest. However, C in octave 9 is available; this is N96. The numbers for each note can be calculated as follows:

C:	<octave>*12
C#,D♭:	1+<octave>*12
D:	2+<octave>*12
D#,E♭:	3+<octave>*12
E:	4+<octave>*12
F:	5+<octave>*12
F#,G♭:	6+<octave>*12
G:	7+<octave>*12
G#,A♭:	8+<octave>*12
A:	9+<octave>*12
A#,B♭:	10+<octave>*12
B:	11+<octave>*12

Using this notation, the command to play an octave of the scale of B flat major is:

```
PLAY"N35N37N39N40N42N44N46N47"
```

This statement will play a chromatic scale (going up in semitones):

```
FOR X=36 TO 48:PLAY"N=X;":NEXT X
```

The **L** command can be used to set the length of the notes. This is followed by a number between 1 and 64. N<*n*> sets the note length to (1/*n*) times the length of a whole note (semibreve). Thus to play crotchets (quarter notes), set the length to L4; to play semiquavers (sixteenth notes), set the length to L16. The default value is L4, so if you do not specify the length of the notes, crotchets will be played.

L commands remain effective until reset. You can set the length of an individual note by following the letter which specifies the note with the appropriate number: A8 will produce a quaver. Note that you cannot do this if the notes are specified using the N command.

Rests can be specified using the **R** command, again followed by a number between 1 and 64. The rests produced are of the same length as notes played using L with the same number, so R1 is a semibreve rest, R4 is a crotchet rest, etc.

Dotted notes or rests can be played by following the command for the note or rest with a full stop. More than one full stop can be used if required; two full stops produce a 'dotted dotted' note.

This one-line program will play 'Londonderry Air':

```
10 PLAY"O4L8EFGL4A.L8GAO5DCO4AGFL4DR8L8FAB-
O5L4 C.L8DCO4FAL2G."
```

If you know the song, you will notice that it is played much too quickly. The tempo can be altered using the T command. T is followed by the number of crotchets to be played per minute, which should be an integer between 32 and 255. The default value is T120. Insert 'T60' at the start of the PLAY string, and the tempo should be about right.

The command **V** can be used to set the volume. The overall volume can be set using the television volume control, but the V command can be used to vary the volume during the piece. The minimum volume is V0, and the maximum is V15.

Substrings can be defined and played using the **X** command. This is followed by the string variable name and a semicolon (as for substrings within DRAW strings).

This program will play 'The Volga Boatmen':

```
10 A$="O4L8GEL4AL2E"
20 B$="O4L4GO5CO4L8B.O5L32CO4BL4A"
30 PLAY"T80V5;XA$;XA$;"
40 PLAYB$
50 PLAYA$+A$
60 PLAY"V7;XA$;"
70 PLAYB$+A$
```

Line 10: defines substring to play the first part of the tune.
Line 20: defines substring to play the second part of the tune.
Line 30: sets the tempo, sets the volume to 5 (fairly soft), and plays the first part of the tune twice.
Line 40: plays the second part of the tune.
Line 50: plays the first part twice more.
Line 60: increases the volume to 7, then plays the first part again.
Line 70: plays the second part then the first part.

The examples given so far have all played just a single melody line. To play three-part harmony, it is necessary to follow the PLAY command with not one but three strings of commands, separated by commas. The first string will be played by channel 1, the second by channel 2 and the third by channel 3.

Remember that a single program line cannot exceed 255 characters. If you are to fit three strings of commands into one line, there will not be much space for each of them, so split the music up into small sections and use a separate line for each section.

If you like you can put the strings of commands into DATA statements, then use a loop to READ and PLAY them in turn. You can then use a standard routine to play any piece of music, changing only the DATA statements to suit the piece being played. If you know in advance how many data items there will be, you can use a FOR...NEXT loop; if not, you can set up a loop using IF...THEN GOTO, with dummy data items marking the end of the list.

Example

```
10 READ P$(1),P$(2),P$(3)
20 IF P$(1)="0" THEN END
30 PLAY P$(1),P$(2),P$(3)
40 GOTO 10
50 DATA "T100V10O4L4FO5D.L8CO4L4B-O5CO4B-
   GFL2D."
60 DATA "T100V8R4O4L2F.E-.L4DL2O3B-."
70 DATA "T100V8R4O2L4B-O3FO4DO3E-GB-O2B-O3FB-
   O2B-"
80 DATA "R4L4FO5D.L8CO4L4B-B-AB-L2O5C.C"
90 DATA "R2L4B-.L8AL4GL2E.E-.F","L4O3FB-
   O2GO3GB-CGO4CO2FO3FAO4E-D"
100 DATA "O4L4FO5D.L8CO4L4B-O5CO4B-
    GFL2D.","R4L2F.E.R1"
110 DATA "L4O4CO2B-O3FO4DO3E-GB-O2B-O3FB-O2B-"
```

```
120 DATA "R4L4FGO5CO4B-AGAL2B-.B-","R4L4DL2E-
    .E-.L4E-CE-L2D"
130 DATA "L4O3FB-E-GO4CO2FO3FO4CL2O2B-
    L4O3FL2O2B-"
140 DATA "0","0","0"
```

Line 10: reads the first (next) three data items and assigns them to P$(1), P$(2) and P$(3).
Line 20: checks to see if the end of the data has been reached.
Line 30: plays P$(1), P$(2) and P$(3) from channels 1, 2 and 3 respectively.
Line 40: returns to line 10.
Lines 50–130: music data.
Line 140: dummy data items to mark the end of the data.

This program plays 'My Bonnie is over the Ocean'. Note that the same tempo (T100) is used for all three parts; the first part is played at a slightly higher volume (V8) than the other two (V6).

You may find that the three parts are slightly out of phase in places. This is because some of the strings contain more commands than others; each command takes time to execute, so the longer strings take longer to play than the short ones. You can remedy this by inserting extra unnecessary commands in the shorter strings (an O4 command when you are already in octave 4, for example), or by putting in short rests.

None of the tunes which have been used as examples so far has contained repeated notes of the same pitch. This is because the computer tends to run notes together: its natural style of playing is legato, not staccato. Try entering:

```
PLAY"O4L4AAAA"
```

You should hear not the four notes that you might expect, but a single long note. To play a succession of short notes, it is necessary to put in short rests, reducing the length of the notes accordingly. To play four crotchet-length As, use:

```
PLAY"O4L8A.R16L8A.R16L8A.R16L8A.R16"
```

or, to make the notes a little longer:

```
PLAY"O4L8A..R32L8A..R32L8A..R32L8A..R32"
```

12.4 THE PLAY FUNCTION

When you RUN a music program, the computer displays the 'Ok' message to say that it has finished and is ready to do something else long before the music stops. This is because the computer contains a special processor to handle the sound; the music data is stored in buffers until it is required by this processor, leaving the main processor free to perform other tasks. You could write a routine to produce animated graphics to accompany the music if you like, or play a tune to fill in time while the computer is performing lengthy calculations. If you are using a signature tune at the start of a program, the time while this is being played can be used to initialize variables and perform other essential preliminary tasks. There may be times, however, when you would like the computer to stop and wait until the music has finished before moving on to the next routine. You could put in a delay loop, adjusting the end value of the counter until the loop is the right length. Alternatively, you can use the function **PLAY (n)** to discover if the music has finished.

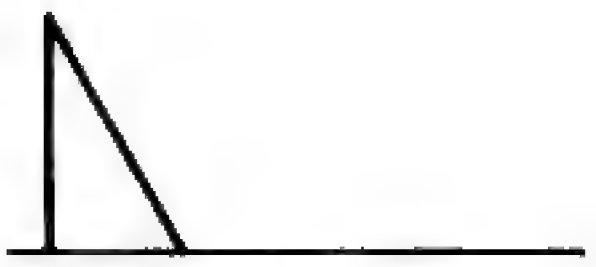
If the PLAY function has an argument of 1, 2 or 3, it will return a value of –1 if the corresponding channel is playing music, and a value of 0 otherwise. PLAY(0) checks all three channels, and returns a value of –1 if any of them are still playing. An IF...THEN statement which checks the value of PLAY(0) can be used to halt the program until the music has finished:

```
100 IF PLAY(0)<>0 THEN 100
```

12.5 VOLUME ENVELOPES

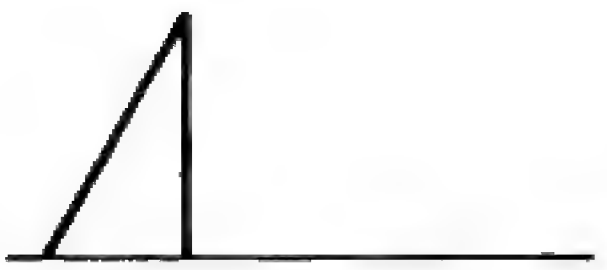
Musical instruments do not produce sound of uniform volume; the volume varies according to a pattern which depends on the nature of the instrument. The sound produced by the computer has volume variations too; there are eight different patterns, or *envelopes*, for you to choose from.

These volume envelopes can be selected within a PLAY string by the **S** command, which is followed by a number between 1 and 15. The patterns available are shown in Figure 8. S1, S2, S3 and S9 all produce the same pattern, as do S4, S5, S6, S7 and S15. The eight different patterns can thus be obtained using S8 to S15.

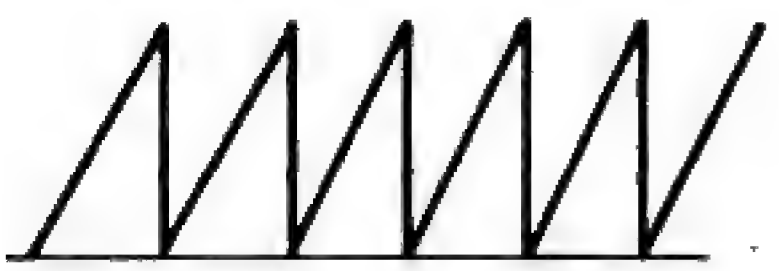


1, 2, 3, 9

11



4, 5, 6, 7, 15

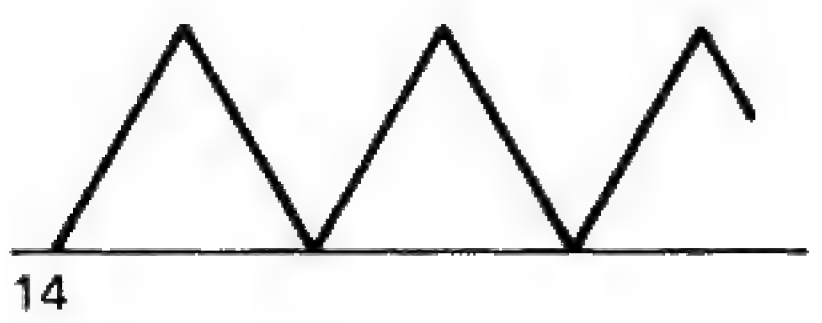


12

8

13

10



14

Figure 8 *Volume envelopes*

The period of the pattern can be set by the **M** (modulation) command. The possible range is from 1 to 65535. The value of M determines the length of the pattern: if M is very large, it is stretched out; if M is small, it is squashed up.

The default values for S and M are 1 and 255 respectively.

Example

```
10 FOR X=8 TO 15
20 FOR N=1 TO 5
30 READ Y
40 CLS:PRINT"S=";X,"M=";Y
50 PLAY"T50V15;S=X;M=Y;O4L16EDL8C.L16CO5L8CCO4
   BL4AL8AGEEL16DCL4D."
```

```
 60 IF PLAY(0)<>0 THEN 60
 70 NEXT N
 80 RESTORE
 90 NEXT X
100 END
110 DATA 10,50,200,500,2000
```

Line 10: sets up a loop to be executed for values of X from 8 to 15 (i.e. for each envelope pattern).
Line 20: sets up a loop to be executed five times.
Line 30: reads next data item (envelope period).
Line 40: clears text screen, prints envelope pattern and period.
Line 50: sets envelope pattern and period to values printed on the screen and plays a short tune.
Line 60: waits until music has stopped.
Line 70: end of inner loop.
Line 80: restores envelope period data (to be used again with next pattern).
Line 90: end of outer loop.
Line 100: end of program.
Line 110: envelope period data.

This program plays a tune using lots of different combinations of values of S and M. The values are displayed on the screen while each version is being played, so you can easily identify the combinations which you like best.

12.6 THE SOUND REGISTERS AND THE SOUND COMMAND

The sound generator processor in the computer has fourteen different registers, the contents of which determine what sound will be produced. When the computer executes a PLAY string, it changes the contents of these registers. It is also possible to change their contents directly, using the **SOUND** command. This is a much more arduous way of programming the computer to produce sound than using PLAY strings, but it is also much more flexible: SOUND commands can be used to produce not only music, but also a wide variety of different sound effects.

The format of the command is:

SOUND <register no.>,<number or expression>

The value of the number or expression must be within the range for the specified register. The functions and data ranges of the registers are given below:

register no.	*function*	*data range*
0	channel A frequency (low byte)	0–255
1	channel A frequency (high byte)	0–15
2	channel B frequency (low byte)	0–255
3	channel B frequency (high byte)	0–15
4	channel C frequency (low byte)	0–255
5	channel C frequency (high byte)	0–15
6	noise frequency	0–31
7	channel selection (tone/noise/off)	0–255
8	channel A volume	0–16
9	channel B volume	0–16
10	channel C volume	0–16
11	envelope period (low byte)	0–255
12	envelope period (high byte)	0–255
13	envelope pattern	0–15

Each register can hold an eight-digit binary number, equivalent to a decimal number between 0 and 255. The ranges of values for the channel frequencies and the envelope period are too great for them to fit into a single register, so two registers are assigned to each of these. The two registers together can hold a sixteen-digit binary number. One register holds the eight rightmost binary digits (the low byte) and the other holds the eight leftmost binary digits (the high byte). To determine the low and high byte values for a number *n*, use the formulae:

low byte = *n*\256
high byte = *n*MOD256
(*n* = 256*(high byte) + (low byte))

Each of the three sound channels can emit either a musical tone or white noise. To produce noise from a channel, you must first set the noise frequency register, then set the channel selection register. To produce a musical note, you must set both channel frequency registers, the channel volume register and, if you want the volume to vary, the envelope selection register and the two envelope period registers, then the channel selection register. You will need at least four, and possibly seven, SOUND commands to produce just one note (plus another

command to switch it off again)!

The number to be assigned to the envelope selection register (register 13) is that used in the PLAY string S command, and the numbers for the envelope period registers (registers 11, 12) are the low and high bytes of the number used in the PLAY string M command (see Section 12.5).

Example

```
SOUND 11,232:SOUND 12,3:SOUND 13,12
```

is equivalent to:

```
PLAY"M1000S12"
```

The channel frequency data figure (registers 0–5) can be calculated from the frequency of the tone which is to be generated, in hertz (Hz), and the clock frequency of the sound generator. The clock frequency for my UK model Sanyo computer is 1996750 Hz; the figure may be different for other MSX computers, so check this in your manual. The formula to be used is:

$$\text{channel frequency data} = \frac{\text{clock frequency}}{\text{output tone frequency (Hz)} * 16}$$

A short program can be used to calculate the high and low bytes of the channel frequency data:

```
10 INPUT"Output tone frequency (Hz)";HO
20 HC=1996750:REM Clock frequency
30 HR=HC/(HO*16)
40 PRINT"Low byte (register 0, 2 or 4) =";HRMOD256
50 PRINT"High byte (register 1, 3 or 5)=";HR\256
60 END
```

Line 10: prints prompt, inputs output frequency.
Line 20: specifies the clock frequency.
Line 30: calculates channel frequency.
Line 40: calculates and prints high byte of channel frequency data.
Line 50: calculates and prints low byte of channel frequency data.
Line 60: end of program.

Example
RUN the above program, inputting a frequency of 440, and you will get:

```
Output tone frequency (Hz)? 440
Low byte (register 0, 2 or 4) = 27
High byte (register 1, 3 or 5= 1
```

So to set the output tone frequency of channel A to 440 Hz, use:

```
SOUND 0,27:SOUND 1,1
```

(You will not hear anything if you execute these commands, because the other registers have not yet been set.)

The noise frequency data (register 6) is similarly related to the clock frequency and the output noise frequency. A reverse calculation can be used to give the output frequency (in hertz) for each of the possible data values:

```
10 PRINT"DATA","OUTPUT FREQUENCY"
20 FOR F%=1 TO 31
30 OF=1996750/ (F%*16)
40 PRINTF%,OF
50 NEXT F%
60 END
```

Line 10: prints column headings.
Line 20: sets up loop, to be executed for values of F% (noise frequency data) from 1 to 31.
Line 30: calculates output frequency.
Line 40: prints noise frequency data and output frequency.
Line 50: end of loop.
Line 60: end of program.

Note that the frequency of the output sound increases as the data value decreases, i.e. low data figures produce high notes, high data figures produce low notes.

The volume of a channel can be set to a minimum by setting the appropriate channel volume register (register 8, 9 or 10) to 0, and to a maximum by setting the register to 15. The numbers from 0 to 15 all produce a constant output volume. To use the volume envelope specified by the contents of registers 11, 12 and 13, you must set the

channel volume register to 16. It is only possible to define one envelope pattern at a time, but this pattern can be used for all three channels if required.

Register 7 is used to switch tone and noise generation on and off for all three channels. Each channel can be set to emit a tone and/or noise, or nothing at all. Bits 7 and 6 of the register (the two leftmost bits) are not used, so should always be set to 0. Bits 5 to 0 are used as shown:

bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
noise	noise	noise	tone	tone	tone
ch. C	ch. B	ch. A	ch. C	ch. B	ch. A

If a bit is set to 0, then its function is turned on, and if it is set to 1, then it is off. To emit noise from channel A, a tone from channel B and nothing from channel C, bits 3 and 1 must be set to 0 and bits 5, 4, 2 and 0 must be set to 1, so the number to be assigned to the register is &B00110101 (decimal 53). To emit noise from channels C and B and a tone from channel C, you must use &B00001011 (decimal 11).

Once the sound has been switched on, it stays on until it is switched off. If you do not put in a statement to switch it off, it will continue even after your program has finished running! You can stop it by pressing <CTRL> and <STOP>, or by resetting or switching off the computer, or by entering the command BEEP.

Examples

(a)
```
10 SOUND 0,27:SOUND 1,1:REM Channel A frequency
20 SOUND 8,15:REM Channel A volume
30 SOUND 7,&B00111110:REM Channel A tone on
40 FOR D=0 TO 500:NEXT D
50 SOUND 7,&B00111111:REM Switch off
60 END
```

This program will emit a 440 Hz tone from channel A.

(b)
```
10 SOUND 9,10:REM Channel B volume
20 SOUND 7,&B00101111:REM Channel B noise on
30 FOR F=31 TO 1 STEP -1
40 SOUND 6,F:REM Noise frequency
50 FOR D=0 TO 10:NEXT D
60 NEXT F
70 SOUND 7,&B00111111:REM Switch off
80 END
```

The frequency of the noise emitted from channel B increases steadily, producing a sort of hissing sound.

(c)
```
10 SOUND 0,208:SOUND 1,3:REM Channel A frequency
20 SOUND 11,232:SOUND 12,3:SOUND 13,14:REM
Envelope
30 SOUND 8,16:REM Channel A envelope on
40 SOUND 9,15:REM Channel B volume
50 SOUND 3,1:REM Channel B high byte
60 SOUND 7,&B00111100:REM Channels A & B tone on
70 SOUND 2,232:REM Channel B low byte
80 FOR D=0 TO 370:NEXT
90 SOUND 2,179
100 FOR D=0 TO 370:NEXT
110 SOUND 2,131
120 FOR D=0 TO 370:NEXT
130 GOTO 70
```

Channel B plays three notes over and over again, while the envelope used for channel A produces a regular pattern of two beats for each note played on Channel B.

Exercise 12.1

Write a program to play the piece of music shown in Figure 9.

Exercise 12.2

Write a program which will play a note of frequency 512 Hz, using envelope pattern 11 and envelope period 1500, for 2 seconds using channel C.

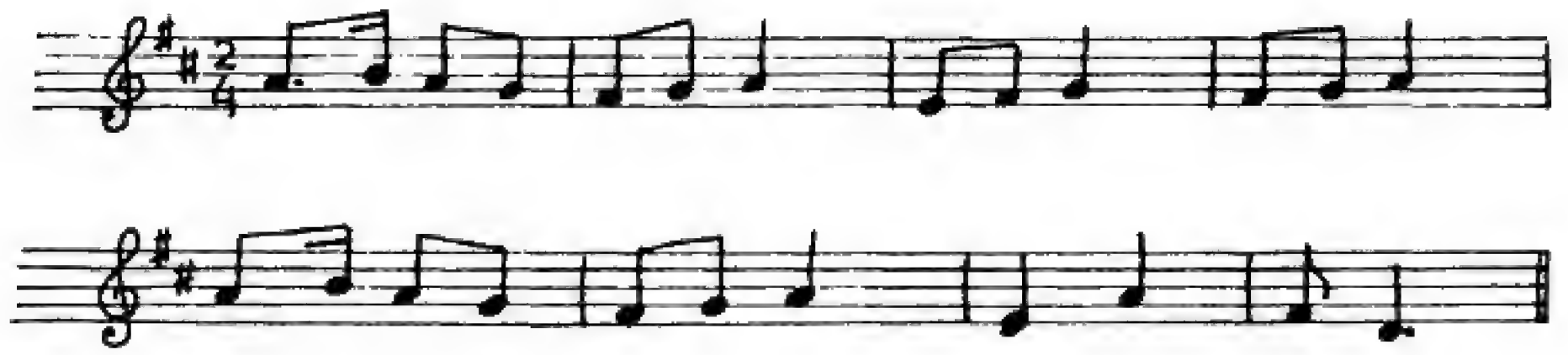


Figure 9

13
File Handling

13.1 WHAT IS A FILE?

When you have written a BASIC program, you can save it for future use on a cassette tape, using the commands and procedure described in Chapter 3. The program is stored as a *program file*. But when you save a program, the contents of the program variables are lost; if, say, you are using an address book program, the program itself is saved but the names and addresses you have entered are not. To save program data, you must use a *data file*.

The computer automatically reserves a section of its memory, the *file control block*, for use as a work area when you are using files. This area contains buffers which are used when writing data to a file, or reading data from a file. The size of the area can be set using the **MAXFILES** command. The format is:

MAXFILES=<number>

The number specified in the MAXFILES command determines how many buffers are available, i.e. how many files can be used at any one time. If it is 0, then only program files, handled by the CSAVE, CLOAD, CLOAD?, SAVE and LOAD commands, can be used. If MAXFILES is set to 1 (the default value), then one data file can be used at a time. The maximum number which can be specified is 15, which enables fifteen files to be used simultaneously (but you are unlikely to need anything like this many).

Cassette data files can be used in much the same way as paper files, to store numeric or character string data. You can label each file with a name for easy identification, store your data in the file, then load it back into the computer.

The file output commands, which are used to write data to the file,

can also be used to print data on the text and graphics screens, and on a printer. Files held on the screen or on a printer are not 'proper' files, as the data cannot be read back into the computer from them. They simply provide another way of printing your data. The ability to set up a graphics screen file and output data to this file is very useful, as it enables you to add text to your graphics. (The normal PRINT commands can only be used to print on the text screens.)

13.2 USING FILES

Before you can use a file, you must open it, using the **OPEN** command. The syntax for this is:

OPEN"<device specifier>:[<file name>]" FOR <mode> AS # <file number>

There are four possible device specifiers:

CAS	cassette tape
CRT	text screen
GRP	graphics screen
LPT	printer

The file name can be omitted; it is best to use one for cassette files, but for screen and printer files it is unnecessary. If you do use a name, it should be not more than six characters long, starting with an alphabetical character.

If the specified device is cassette tape, then the mode can be OUTPUT (data is to be written to the file) or INPUT (data is to be read from the file). With the other devices, only the OUTPUT mode is available.

The file number determines which buffer in the file control block is to be used. It must be between 1 and the number specified in the MAXFILES statement.

Example

```
10 MAXFILES=2
20 OPEN"CAS:FILE1" FOR OUTPUT AS #1
30 OPEN"GRP:" FOR OUTPUT AS #2
```

Two files have been opened, a cassette file named FILE1 to which data

is to be written, and a graphics screen file to enable text to be printed on the graphics screen.

The commands **PRINT #** and **PRINT # USING** can be used to write data to a file which has been opened for output. The syntax to use for these commands is:

```
PRINT #<file number>,<data>
PRINT #<file number> USING <format>;<data>
```

The file number must be the same as the number used when the file was opened. The formats available with PRINT # USING are the same as those available with PRINT USING (see Section 5.2).

If several items of string data are being output with one PRINT # command, you must insert "," between the items to separate them. Otherwise when the data is read back into the computer from the file, the items may be run together. There is no need to do this with numeric data. The computer automatically adds a carriage return and line feed to punctuate the data at the end of a PRINT # statement.

When you have finished writing data to the file, use a **CLOSE** statement to close the file and free the file buffer for use with another file:

```
CLOSE #<file number>
```

This also puts a marker at the end of the file, so you can tell when all the data has been read (see below).

Example

```
10 MAXFILES=2
20 FOR C=1 TO 5
30 INPUT "Enter data item";A$(C)
40 NEXT C
50 PRINT "Make sure the cassette recorder is ready, and set to
   RECORD"
60 PRINT "Press any key to continue.":X$=INPUT$(1)
70 OPEN"CAS:TEST" FOR OUTPUT AS #1
80 FOR C=1 TO 5
90 PRINT #1,A$(C);",";
100 NEXT C
110 CLOSE #1
120 PRINT "Data filed. Now stop the cassette recorder."
130 END
```

Line 10: sets maximum number of files to two.
Line 20: sets up loop to be executed five times.
Line 30: prints prompt, inputs data item.
Line 40: end of loop.
Line 50: prints prompt.
Line 60: prints prompt, waits for keypress.
Line 70: opens cassette file named TEST for output using buffer 1.
Line 80: sets up loop, to be executed five times.
Line 90: outputs data item, followed by a comma, using buffer 1.
Line 100: end of loop.
Line 110: closes the file.
Line 120: prints prompt.
Line 130: end of program.

This program will input five data items, and write them to a cassette file named TEST. Note the inclusion of prompts to remind the user to switch the cassette recorder on and off.

To read in data stored in a cassette file, you must first OPEN the file in INPUT mode. Use the file name you used when filing the data, or omit the file name to read the contents of the first file on the tape. Then use **INPUT #** or **LINE INPUT #** to read in the data. The syntax is:

INPUT #<file number>,<numeric or string variable>
LINE INPUT #<file number>,<string variable>

INPUT # can be used to read in numeric data, or string data separated by commas, line feeds or carriage returns. LINE INPUT # is used to read in string data separated by carriage returns only, i.e. the data items may include commas.

The **EOF** (end of file) function, with the file number as its argument, can be used to check if the last item of data in the file has been read. It returns a value of –1 if the end of the file has been reached, 0 otherwise.

Example
This program will read the file created by the last program, and display its contents on the screen.

```
10 MAXFILES=2
20 PRINT "Rewind the cassette, and set to PLAY"
30 PRINT "Press any key to continue.":X$=INPUT$(1)
40 OPEN"CAS:TEST" FOR INPUT AS #2
```

```
 50 IF EOF(2)=-1 THEN GOTO 90
 60 INPUT #2,A$
 70 PRINT A$
 80 GOTO 50
 90 CLOSE #2
100 END
```

Line 10: sets the maximum number of files to two.
Line 20: prints prompt.
Line 30: prints prompt, waits for keypress.
Line 40: opens cassette file named TEST for input using buffer 2.
Line 50: jumps to line 90 if end of file has been reached.
Line 60: inputs data item using buffer 2.
Line 70: prints data item on screen.
Line 80: returns to line 50.
Line 90: closes file.
Line 100: end of program.

13.3 PRINTING ON THE GRAPHICS SCREEN

To print on the graphics screen, you must open an OUTPUT file to the graphics screen, then write the text to be printed to this file using PRINT #.

As there is no text cursor on the graphics screen, the printing will start at the current position of the graphics cursor. You can use a BM command within a DRAW string to position the cursor before printing. The text is carried over from one line to the next in the normal way.

Example

```
 10 SCREEN 2:COLOR 1,15,15:CLS
 20 OPEN"GRP:" FOR OUTPUT AS #1
 30 CIRCLE(80,70),50,8:PAINT(80,70),8
 40 DRAW"BM60,70":PRINT #1,"Circle"
 50 LINE(150,20)-(250,120),7,BF
 60 DRAW"BM180,70":PRINT #1,"Square"
 70 CLOSE #1
 80 GOTO 80
```

Line 10: selects graphics mode 2, foreground black, background and border white, and clears the screen.

Line 20: opens graphics screen file for output using buffer 1.
Line 30: draws and paints a red circle, centre (80,70), radius 50.
Line 40: moves graphics cursor to (60,70), prints 'Circle' using file buffer 1.
Line 50: draws a solid blue square, top left-hand corner (150,20).
Line 60: moves graphics cursor to (180,70), prints 'Square' using file buffer 1.
Line 70: closes file.
Line 80: holds the display on the screen.

The word 'Circle' is printed inside a red circle, and the word 'Square' is printed inside a blue square.

The text is printed in the current foreground colour. As this colour can be changed by a COLOR statement, you can produce multicoloured text if you wish (something which is not possible with the normal PRINT commands).

Example

```
 10 SCREEN 2:COLOR 1,15,15:CLS
 20 OPEN"GRP:" FOR OUTPUT AS #1
 30 DRAW"BM80,100"
 40 FOR C=1 TO 13
 50 READ A
 60 COLOR C:PRINT #1,CHR$(A);
 70 NEXT C
 80 CLOSE #1
 90 GOTO 90
100 DATA 77,85,76,84,73,67,79,76,79,85,82,69,68
```

Line 10: selects graphics mode 2, foreground black, background and border white, and clears the screen.
Line 20: opens graphics screen file for output using buffer 1.
Line 30: moves graphics cursor to (80,100).
Line 40: sets up loop to be executed for values of C from 1 to 13.
Line 50: reads next data item.
Line 60: sets foreground colour to C, prints character using file buffer 1.
Line 70: end of loop.
Line 80: closes file.
Line 90: holds the display on the screen.
Line 100: data items: ASCII codes for characters to be printed.

The word 'MULTICOLOURED' is printed, using a different colour for each letter.
Note that the start X co-ordinate (line 30) is divisible by 8. Each letter is eight pixels wide; if the start co-ordinate was not divisible by 8, the letters would not all be different colours because of the smudging effect in mode 2.

In graphics mode 2 the text appears normal size, but in mode 3 it is much larger than normal. This is because in mode 3 the characters are made up of 4×4 pixel blocks instead of single pixels.

Example

```
10 SCREEN 3:COLOR 10,12,12:CLS
20 OPEN"GRP:" FOR OUTPUT AS #1
30 DRAW"BMO,50"
40 PRINT #1," BIG":PRINT #1," LETTERS"
50 CLOSE #1
60 GOTO 60
```

Line 10: selects graphics mode 3, foreground dark yellow, background and border dark green, and clears the screen.
Line 20: opens graphics screen file for output using buffer 1.
Line 30: moves graphics cursor to (0,50).
Line 40: prints 'BIG' and 'LETTERS' using file buffer 1.
Line 50: closes file.
Line 60: holds display on the screen.

Exercise 13.1

Write a program which will input ten numbers and assign them to the variables X(1) to X(10), then save them in a cassette file named NUM.

Exercise 13.2

Write a program which will read the ten numbers in the file NUM back into the variables X(1) to X(10).

Exercise 13.3

Write a program which will print a table showing the colour codes from 1 to 14 and the corresponding colours on the mode 2 screen, each line being printed in the appropriate colour. (Use a white background.)

14
Sprite Graphics

14.1 SPRITE SCREENS AND SIZES

Sprites are special *user-defined characters* which can be moved around the screen without affecting the background. Normally, when you put a character on the screen, whatever previously occupied the position in which you put the character is obliterated. With sprites, this does not happen. It is as if the sprites are placed not on the screen itself, but on transparent sheets – *sprite planes* – which are placed in front of the screen. The MSX specification provides 32 of these sprite planes, numbered from 0 (the top plane) to 31 (the back plane, closest to the screen). One sprite can be displayed on each of these planes, and each sprite can pass in front of the sprites on higher-numbered planes as well as passing across the background picture without disrupting it. Figure 10 shows the configuration of the background display screen and the sprite planes.

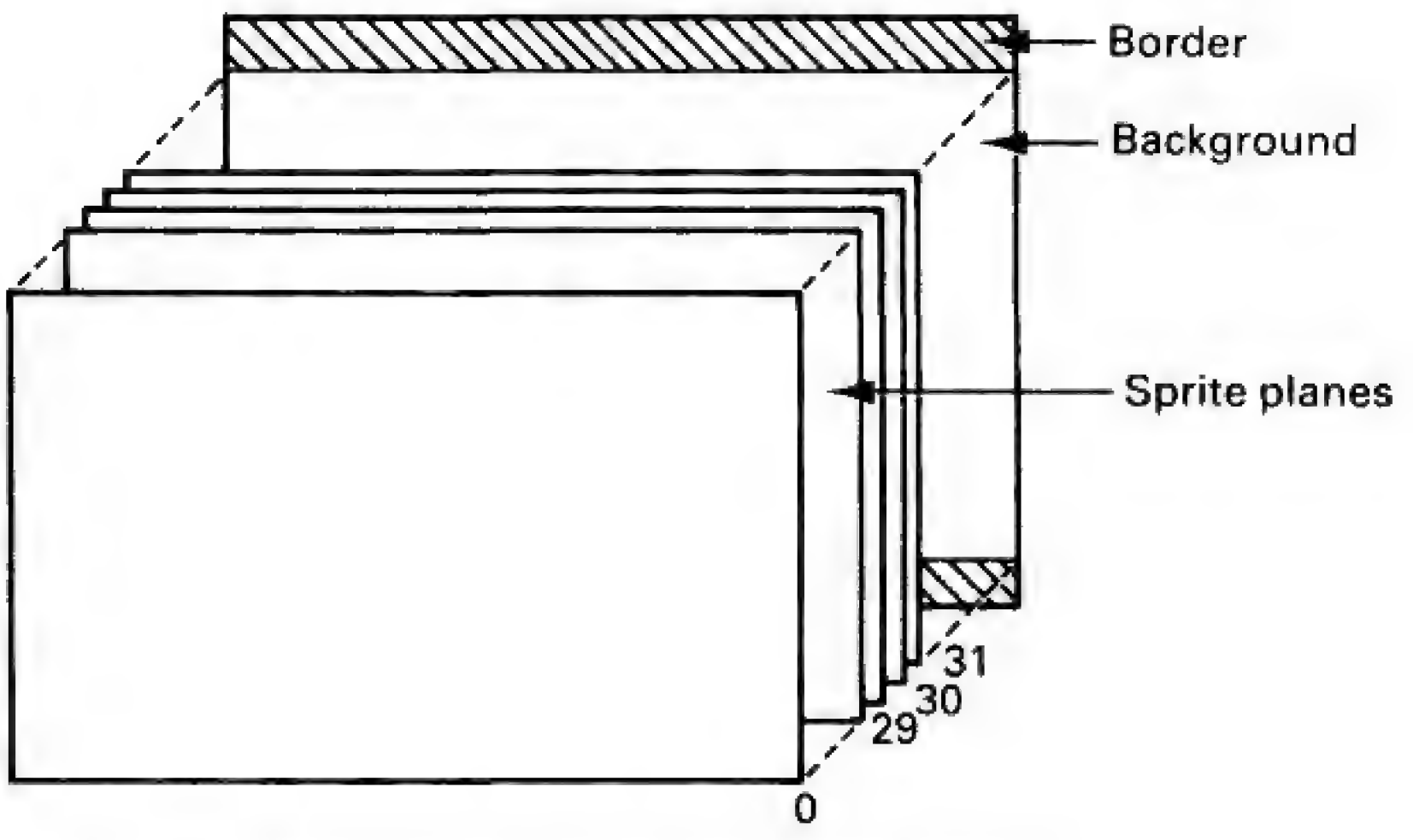


Figure 10 *The background screen and sprite planes*

Sprites can be used in text mode 1, graphics mode 2 and graphics mode 3, but not in text mode 0. Two different sizes are available, 8×8 dots and 16×16 dots; they can be displayed unmagnified, when each dot corresponds to one pixel on the screen, or magnified, when each dot corresponds to a 2×2 pixel square. The size and magnification are set by the second parameter of the **SCREEN** command, which can take the following values:

0	8×8 dots, unmagnified
1	8×8 dots, magnified
2	16×16 dots, unmagnified
3	16×16 dots, magnified

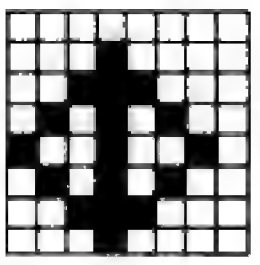
Thus SCREEN 2,3 selects graphics mode 2 with 16×16 dot magnified sprites; SCREEN 1,0 selects text mode 1 with 8×8 dot unmagnified sprites. Figure 11 shows the four different sprite size options.

The sprite size selected with the SCREEN command applies to all 32 sprite planes; you cannot use magnified sprites on some planes, and unmagnified sprites on others. You can, however, display sprites of differing sizes if you select the larger (16×16) size. Each sprite is composed of a pattern of dots in the sprite colour, and transparent dots which allow the background (or higher-numbered sprites) to show through. Using the 16×16 sprite size, you can define sprites which will appear small by setting most of the dots to transparent.

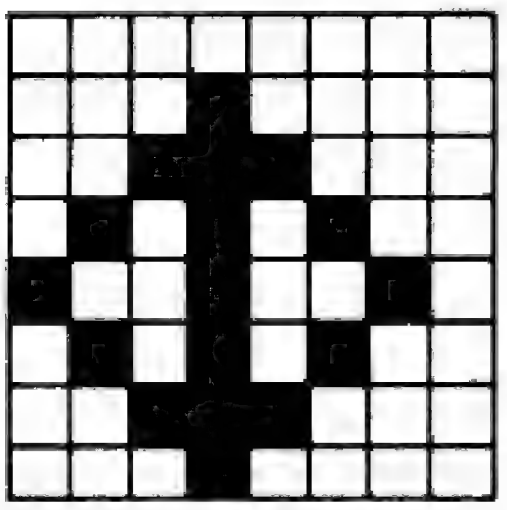
MSX sprites have only one colour each (plus transparent), but you can produce 'multicoloured sprites' from two or more single-colour sprites on top of one another, by designing them so that some of the transparent dots in the top sprite correspond to coloured dots in the sprite underneath. The component sprites of your 'multicoloured sprite' must, of course, be moved around the screen together.

14.2 DESIGNING AND DEFINING SPRITE PATTERNS

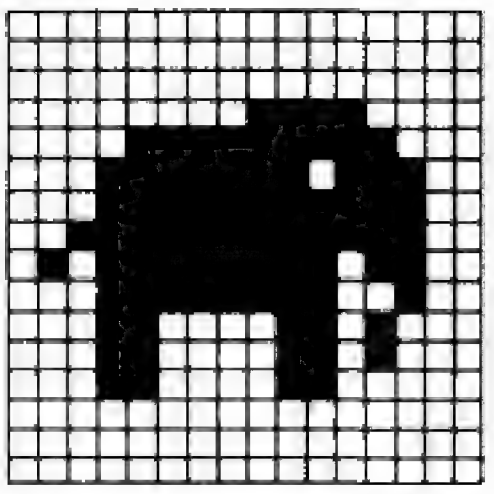
Sprites can be designed using either paper and a pencil, or a special sprite designer program (there is one listed in Chapter 19 of this book). You start with a pattern of empty little squares – eight rows of eight squares or sixteen rows of sixteen squares, depending on the size you have selected – and fill squares in until the pattern looks right. If you are using paper and pencil, it is a good idea to use ink for the pattern of



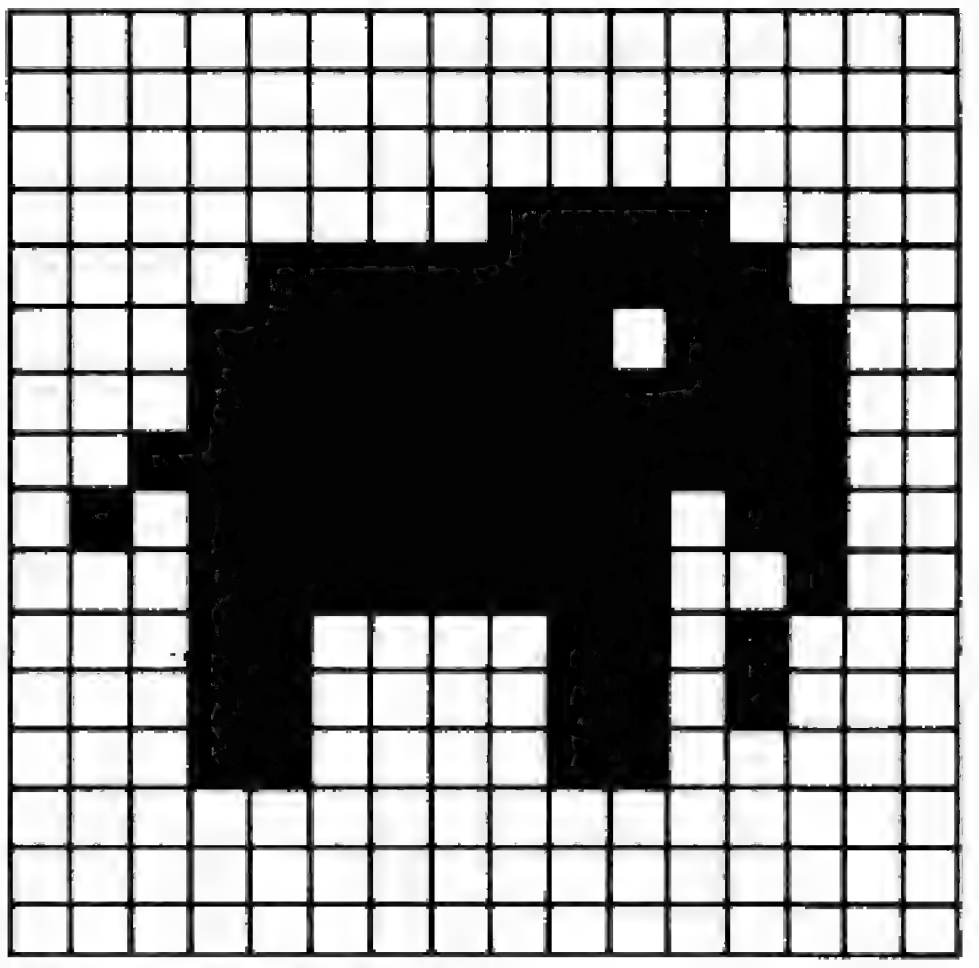
8 × 8 unmagnified



8 × 8 magnified



16 × 16 unmagnified



16 × 16 magnified

Figure 11 *Sprite sizes*

squares you start with and pencil to fill them in; then if you change your mind about filling a square in, you can easily erase its contents without also erasing the square itself.

Once you have produced a satisfactory design, this pattern of empty and filled squares must be transformed into numeric data to be fed into the computer. For an 8×8 sprite, the contents of each row of eight squares can be represented by an eight-digit binary number, with a filled square corresponding to a 1 and an empty square to a 0. The data for the whole sprite consists of eight binary numbers, one for each row of squares. The numbers can be fed into the computer in binary form, or converted to decimal, when they will be easier and quicker to enter in your program. (If you are using the sprite designer program, this will do all the hard work for you and present the data to you in decimal form.)

Here is a sprite pattern (using asterisks to represent filled squares and – signs for empty squares), and the corresponding data:

---*----	&B00010000	16
--***---	&B00111000	56
-*-*-*--	&B01010100	84
----*-	&B10010010	146
---*----	&B00010000	16
---*----	&B00010000	16
---*----	&B00010000	16
---*----	&B00010000	16

The command used to define sprite patterns is **SPRITE$**. It takes the form:

SPRITE$(<sprite pattern no.>) = <string>

If you are using 8×8 sprites, the sprite pattern number can be any number between 0 and 255; there is enough memory space available in the VRAM for 256 different 8×8 sprite patterns. The string consists of the eight characters whose ASCII codes are equal to the data numbers derived from the sprite pattern, starting with the number corresponding to the topmost row of squares. You can form the string using CHR$() functions, or use the actual characters if you know what they are.

The pattern shown above can be defined by:

```
SPRITE$(1)=CHR$(16)+CHR$(56)+CHR$(84)
   +CHR$(146)+CHR$(16)+CHR$(16)+CHR$(16)+CHR$(16)
```

Note: the sprite pattern number has been arbitrarily chosen. It could be any number between 0 and 255.

Alternatively, you can put the data in a DATA statement, and use a FOR . . . NEXT loop to READ it and add the appropriate characters to the string:

```
10 SCREEN 2,0
20 S$=""
30 FOR C=1 TO 8
40 READ X
50 S$=S$+CHR$(X)
60 NEXT C
70 SPRITE$(1)=S$
80 DATA 16,56,84,146,16,16,16,16
```

Line 10: selects graphics mode 2, with 8×8 unmagnified sprites.

Line 20: initializes string.
Line 30: sets up loop, to be executed eight times.
Line 40: reads next data item and assigns it to variable X.
Line 50: adds appropriate character to string.
Line 60: end of loop.
Line 70: defines sprite pattern.
Line 80: sprite data.

You will not see anything happen if you RUN this routine; the sprite has been defined, but has not yet been displayed on the screen.

Size 16×16 sprites are treated as four 8×8 sprites, arranged as shown:

```
1    3
2    4
```

To define a 16×16 sprite, the string in the SPRITE$ statement must contain 32 characters, eight characters for each of these 8×8 sprites. The sprite pattern number must be a number between 0 and 63, as there is room for only 64 of these larger patterns.

Here is a 16×16 sprite pattern:

```
--------*--------
-------***-------
------*****------
-----*******-----
----*********----
---***********---
--*************--
-***************-
****************
-***************-
--*************--
---***********---
----*********----
-----*******-----
------*****------
-------***-------
```

The 8×8 sprites of which this is made up are (with their data):

	binary	*decimal*
sprite 1		
--------	&B00000000	0
-------*	&B00000001	1
------**	&B00000011	3
-----***	&B00000111	7
----****	&B00001111	15
---*****	&B00011111	31
--******	&B00111111	63
-*******	&B01111111	127
sprite 2		
--------	&B00000000	0
-*******	&B01111111	127
-*******	&B01111111	127
--******	&B00111111	63
---*****	&B00011111	31
--------	&B00000000	0
--------	&B00000000	0
--------	&B00000000	0
sprite 3		
*-------	&B10000000	128
------	&B10100000	160
*-**----	&B10110000	176
*-***---	&B10111000	184
*-*****-	&B10111100	188
*-******-	&B10111110	190
*-*******	&B10111111	191
*-------	&B10000000	128
sprite 4		
--------	&B00000000	0
********	&B11111111	255
********	&B11111111	255
*******-	&B11111110	254
******--	&B11111100	252
--------	&B00000000	0
--------	&B00000000	0
--------	&B00000000	0

This routine defines the 16×16 sprite:

```
  5 REM Define yacht sprite
 10 SCREEN 2,3:COLOR 8,5,5:CLS
 20 S$=""
 30 FOR C=1 TO 32
 40 READ X
 50 S$=S$+CHR$(X)
 60 NEXT C
 70 SPRITE$(2)=S$
 80 DATA 0,1,3,7,15,31,63,127
 90 DATA 0,127,127,63,31,0,0,0
100 DATA 128,160,176,184,188,190,191,128
110 DATA 0,255,255,254,252,0,0,0
```

This is the same as the routine given for 8×8 sprites, except for these lines:

Line 5: comment (ignored by the BASIC interpreter).
Line 10: selects graphics mode 2, with 16×16 magnified sprites, foreground colour medium red, background and border light blue, and clears the screen.
Line 30: sets up loop, to be executed 32 times.
Lines 80–110: sprite data.

Again, nothing will appear to happen if you RUN this routine (apart from the blue graphics screen appearing momentarily). A routine to display and move this sprite is given in Section 14.3.

14.3 DISPLAYING AND MOVING SPRITES

There are 32 different sprite planes, numbered from 0 (the frontmost plane) to 31; a single sprite can be displayed on each of these planes, so you can have up to 32 sprites visible at any one time. You can define up to 255 different sprite patterns if you are using 8×8 sprites (64 patterns for 16×16 sprites), and any of these patterns can be used on any of the sprite planes. You can use the same pattern on all 32 planes, giving you 32 sprites of the same pattern (but not necessarily of the same colour – see below), or you can use all the patterns in turn on one of the planes.

The command to display a sprite pattern on one of the sprite planes

is **PUT SPRITE.** The syntax used is:

PUT SPRITE <sprite plane no.>[,(co-ordinates>)],
[<colour code>][,<sprite pattern no.>]

The sprite plane number must be between 0 and 31.

If co-ordinates are given, then these specify the position of the top left-hand corner of the sprite. The co-ordinates can be specified using numbers, variables or expressions. Absolute or relative co-ordinates can be used, and the graphics cursor is moved to the specified position. If no co-ordinates are specified, then the sprite is placed with its top left-hand corner at the current position of the graphics cursor.

The colour code determines the colour of the sprite foreground. The sprite background is always transparent. If no colour is specified, then the current foreground colour is used.

If no sprite pattern number is specified, this is assumed to be the same as the sprite plane number.

When a PUT SPRITE command is used, any sprite which was previously displayed on the specified sprite plane is automatically erased. You can therefore move a sprite around the screen by using repeated PUT SPRITE commands, with the same sprite plane, pattern and colour numbers but different co-ordinates.

This routine will move the yacht sprite defined in Section 14.2 across the screen:

```
200 REM Move yacht sprite
210 FOR X=-16 TO 255
220 PUT SPRITE 0, (X,100),8,2
230 FOR D=0 TO 50:NEXT D
240 NEXT X
250 END
```

Line 200: comment.
Line 210: sets up loop to be executed for each value of X from -16 to 255.
Line 220: puts previously defined sprite (pattern 2) on sprite plane 0, in medium red (colour 8), at the co-ordinates (X,100).
Line 230: delay loop, slows down the movement of the sprite.
Line 240: end of loop.
Line 250: end of program.

Type in the yacht sprite definer routine together with this routine. When

you RUN the whole program you should see a red yacht move smoothly across the screen from left to right.

The sprite planes are slightly larger than the display screen, with the X co-ordinates running from –32 to 255, and the Y co-ordinates from –32 to 191. These margins at the top and left-hand side of the screen allow you to move sprites smoothly on and off the edges of the screen. The co-ordinates specified in the PUT SPRITE command can be anywhere within the integer range (–32768 to 32767), and there is a wraparound facility, so sprites which are moved off the right-hand edge of the screen reappear at the left, and sprites which are moved off the bottom reappear at the top. Try running this program:

```
10 REM Sprite wraparound demo.
20 SCREEN 2,0:COLOR 15,1,1:CLS
30 DRAW"BM104,104;NE4NF4NG4NH4"
40 SPRITE$(0)=STRING$(8,255)
50 FOR C=-280 TO 500
60 PUT SPRITE 0,(C,100),2,0
70 PUT SPRITE 1,(100,C),4,0
80 FOR D=0 TO 10:NEXT D
90 NEXT C
100 END
```

Line 10: comment.
Line 20: selects graphics mode 2 with 8×8 unmagnified sprites, foreground white, background and border black, and clears the screen.
Line 30: draws a white cross on the screen, with its centre at the point (104,104).
Line 40: defines sprite pattern 0 as a solid square (255=&B11111111).
Line 50: sets up loop to be executed for each value of C from –280 to 500.
Line 60: puts a medium green square sprite on sprite plane 0 at co-ordinates (C,100).
Line 70: puts a dark blue square sprite on sprite plane 1 at co-ordinates (100,C).
Line 80: delay loop, to slow down movement of sprites.
Line 90: end of loop.
Line 100: end of program.

A white cross is drawn on a black screen. Then two sprites start to come into view, a green sprite, at the left of the screen and a blue sprite at the

top. These move smoothly across and down the screen respectively. They cross over in front of the white cross, the blue sprite passing behind the green sprite, which has priority as it is on a lower numbered plane. The blue sprite disappears off the bottom of the screen, and shortly afterwards the green sprite disappears off the right-hand side. After a short delay the two sprites reappear and move over the screen again, crossing over at the same point as before. This time there is no delay between the green sprite disappearing at the right of the screen and the two sprites reappearing. The sprites cross over in front of the white cross for a third time.

You will see from this that when a sprite is moved off the top or bottom of the screen, there is always a delay before it reappears at the bottom or top respectively, but when a sprite is moved off the edge of the screen it reappears immediately at the opposite edge if the X co-ordinate is positive. If you want to move a sprite off the edge of the screen without it reappearing at the opposite edge, use negative X co-ordinates.

This program will display five square sprites in different colours:

```
10 SCREEN 2,1:COLOR 1,15,15:CLS
20 SPRITE$(0)=STRING$(8,255)
30 PUT SPRITE 1,(100,0),2,0
40 PUT SPRITE 2,(100,20),4,0
50 PUT SPRITE 3,(100,40),6,0
60 PUT SPRITE 4,(100,60),8,0
70 PUT SPRITE 5,(100,80),10,0
80 GOTO 80
```

Line 10: selects graphics mode 2 with 8×8 magnified sprites, foreground black, background and border white, and clears the screen.
Line 20: defines sprite pattern 0 as a solid square.
Line 30: displays a green square sprite at (100,0).
Line 40: displays a blue square sprite at (100,20).
Line 50: displays a dark red square sprite at (100,40).
Line 60: displays a medium red square sprite at (100,60).
Line 70: displays a yellow square sprite at (100,80).
Line 80: holds the picture on the screen.

RUN this program to make sure that all five sprites are displayed as they should be, then press <CTRL> and <STOP> to break out of the loop in line 80, and edit line 50 to set the Y co-ordinate of the dark red sprite to 208:

```
50 PUT SPRITE 3,(100,208),6,0
```

When you RUN the program again, only the first two sprites will appear! Setting the Y co-ordinate of a sprite to 208 has the rather odd effect of disabling all the sprites on higher numbered sprite planes.

Edit the program again so that all five sprites have a Y co-ordinate of 100, but their X co-ordinates are 20, 40, 60, 80 and 100 respectively. Now when you RUN the program, four sprites will appear instead of five! The sprite on plane 5 is missing. Change line 70 to:

```
70 FOR Y=60 TO 140:PUT SPRITE 5, (100,Y),10,0:FOR D=0
   TO 20:NEXT D,Y
```

Now you will see that the sprite on plane 5 is not displayed when it is on the same horizontal line as the other four sprites. You cannot display more than four sprites on one horizontal line; if you attempt to do so, only the four sprites on the lowest-numbered sprite planes will be displayed.

14.4 EXTRA LARGE, MULTICOLOURED AND ANIMATED SPRITES

The MSX specification places strict limitations on the size of sprites, and allows you to use only one colour per sprite. However, you can simulate outsized sprites and multicoloured sprites by using two or more ordinary sprites which are moved around the screen together. Multicoloured sprites can be produced by displaying several sprites of different colours at the same co-ordinates, and large sprites can be produced by displaying two or more ordinary sprites next to one another.

Remember that you can only display four ordinary sprites on any horizontal screen line. If you are using multicoloured sprites which are each made up of two ordinary sprites, then obviously you will only be able to display two of these on a line. With three- and four-coloured sprites, only one can be displayed on each line. It is impossible to produce multicoloured sprites using more than four different colours.

When you place one sprite on top of another, the colours of the first sprite may appear to flash. This effect can be minimized by placing the darker-coloured sprite on the screen first.

The program below defines and moves a black and yellow bee sprite. The SOUND statements produce a buzzing sound!

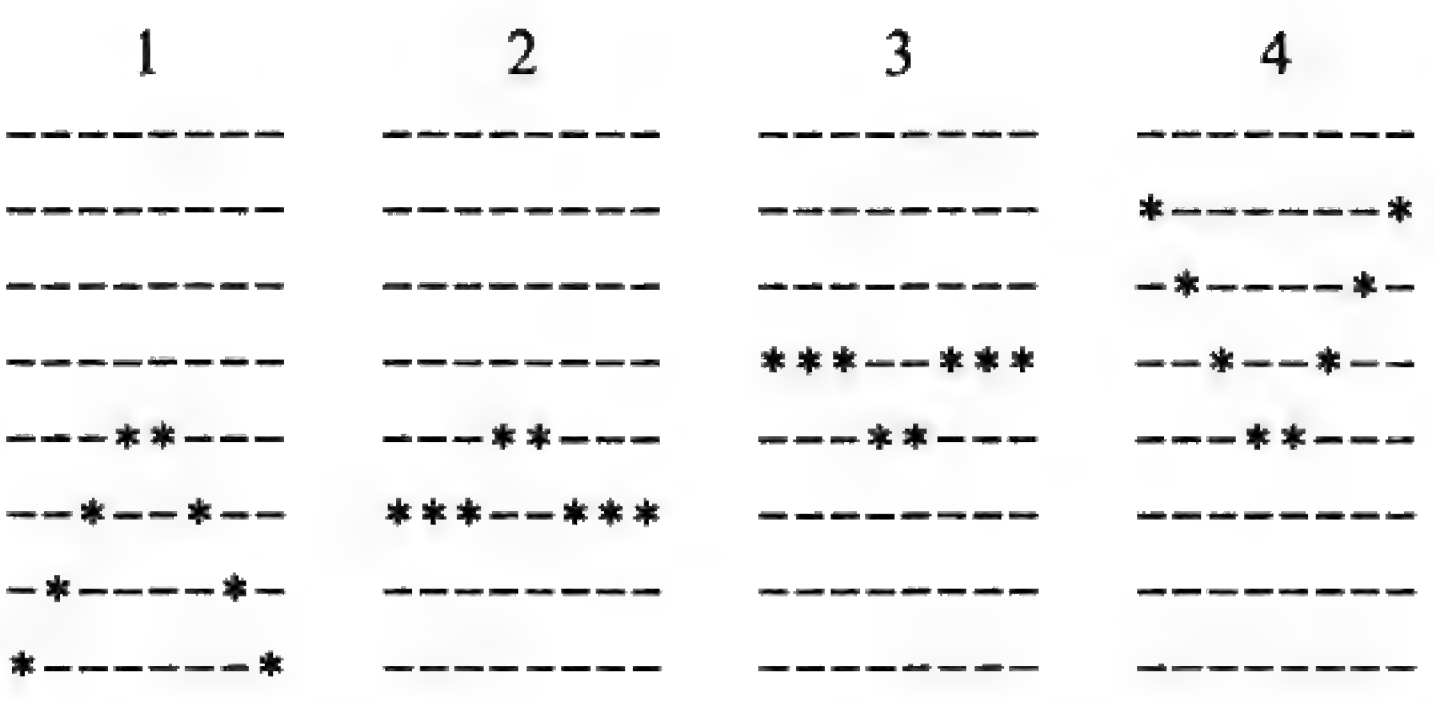
```
 10 REM Buzzing bee
 20 SCREEN 2,3:COLOR 1,7,7:CLS
 30 FOR S=0 TO 1
 40 S$=" "
 50 FOR C=1 TO 32
 60 READ X
 70 S$=S$+CHR$(X)
 80 NEXT C
 90 SPRITE$(S)=S$
100 NEXT S
110 DATA 0,34,17,17,17,42,170,170,170,170,42,17,17,17,34,0
120 DATA 0,0,0,0,0,153,190,188,188,190,153,0,0,0,0,0
130 DATA 0,0,0,0,0,85,85,85,85,85,85,0,0,0,0,0
140 DATA 0,0,0,0,0,0,64,64,64,64,0,0,0,0,0,0
150 X=0:Y=100
160 R=RND(-TIME)
170 SOUND 0,100:SOUND 1,0:SOUND 8,16
180 SOUND 11,200:SOUND 12,0:SOUND 13,8
190 SOUND 7,&B00111110
200 R=RND(1):IF R<0.5 THEN D=-1 ELSE D=1
210 Y=Y+D
220 IF Y<0 THEN Y=0
230 IF Y>160 THEN Y=160
240 X=X+1
250 PUT SPRITE 0,(X,Y),1
260 PUT SPRITE 1,(X,Y),11
270 GOTO 200
```

Line 10: comment.
Line 20: selects graphics mode 2 with 16×16 magnified sprites, foreground black, background sky blue, and clears screen.
Line 30: sets up loop to be executed twice (once for each sprite pattern).
Line 40: initializes string.
Line 50: sets up loop to be executed 32 times (once for each data item).
Line 60: reads next data item and assigns it to variable X.
Line 70: adds appropriate character to string.
Line 80: end of inner loop.
Line 90: defines sprite pattern S.
Line 100: end of outer (S) loop.

Lines 110,120: data for black sprite.
Lines 130,140: data for yellow sprite.
Line 150: sets initial co-ordinates for sprites.
Line 160: randomizes RND function.
Line 170: sets tone frequency and volume for channel A.
Line 180: sets envelope pattern and period.
Line 190: switches channel A tone on.
Line 200: decides randomly whether Y co-ordinate is to be incremented or decremented.
Line 210: increments or decrements Y co-ordinate of sprites.
Line 220: ensures top of sprites will not be above top of screen.
Line 230: ensures bottom of sprites will not be below bottom of screen.
Line 240: increments X co-ordinate of sprites.
Line 250: displays black sprite.
Line 260: displays yellow sprite.
Line 270: returns to line 200.

Designing and defining extra large sprites is straightforward, but when you are displaying them, remember that magnified sprites occupy more screen space than unmagnified ones. If you want to place two 8×8 magnified sprites beside each other, with the sprite on the left at co-ordinates (X,Y), then the co-ordinates of the sprite on the right should be (X+16,Y). Similarly, to place two 16×16 magnified sprites one beneath the other, if the co-ordinates of the top sprite are (X,Y) then the co-ordinates of the bottom sprite should be (X,Y+32).

Animation can be achieved by displaying several different sprite patterns in rotation. For example, to show a bird flapping its wings you must define sprite patterns showing the bird with its wings in several different positions. The best effect will obviously be produced by using a large number of patterns, each differing only slightly from the one before. Here just four different patterns will be used to illustrate the principle. The patterns are:



To make the 'bird' flap its wings up and down, these patterns must be displayed in the order 1, 2, 3, 4, 3, 2. For ease of programming, six sprite patterns will be defined, with pattern 5 the same as pattern 3 and pattern 6 the same as pattern 2. Then the patterns can be displayed in the order 1, 2, 3, 4, 5, 6.

Here is the program to display the bird flapping its wings:

```
 10 REM Animated bird
 20 SCREEN 1,1:COLOR 1,7,7:CLS
 30 FOR S=1 TO 6
 40 S$=""
 50 FOR C=1 TO 8:READ X:S$=S$+CHR$(X):NEXT C
 60 SPRITE$(S)=S$
 70 NEXT S
 80 DATA 0,0,0,0,24,36,66,129,0,0,0,0,24,231,0
 90 DATA 0,0,0,231,24,0,0,0,0,129,66,36,24,0,0,0
100 DATA 0,0,0,231,24,0,0,0,0,0,0,0,24,231,0,0
110 FOR S=1 TO 6
120 PUT SPRITE 0,(120,90),1,S
130 FOR D=0 TO 20:NEXT D
140 NEXT S
150 GOTO 110
```

Line 10: comment.
Line 20: selects text mode 1 with 8×8 magnified sprites, foreground black, background and border sky blue, and clears the screen.
Line 30: sets up loop to be repeated six times (once for each sprite pattern).
Line 40: initializes string.
Line 50: reads eight data items and adds appropriate characters to string.
Line 60: defines sprite pattern.
Line 70: end of loop.
Line 80: data for sprite patterns 1 and 2.
Line 90: data for sprite patterns 3 and 4.
Line 100: data for sprite patterns 5 and 6.
Line 110: sets up loop to be repeated six times (once for each sprite pattern).
Line 120: displays appropriate sprite pattern on sprite plane 0, at co-ordinates (120,90), in black.
Line 130: delay loop (to slow down animation).
Line 140: end of loop.
Line 150: returns to line 110.

When you stop the bird program (by pressing <CTRL> and <STOP>), you will notice that the bird sprite remains on the screen. This is because the sprite was displayed in text mode 1, not in one of the two graphics modes. You can get rid of it by entering a new SCREEN command, or a PUT SPRITE command which sets its Y co-ordinate to 209.

14.5 USING STATIC SPRITES AS PART OF THE BACKGROUND SCREEN DISPLAY

Although sprites are designed to be easily moveable, static sprites can also be a valuable tool. They can be used to enhance the quality of your graphics even in programs in which no movement is required.

Static sprites are usually placed on the higher-numbered sprite planes (planes 30, 31), so that moveable sprites (on lower-numbered planes) will pass in front of them. However, you can also place static sprites on the lowest-numbered planes, to produce a three-dimensional effect: they can represent fixed objects which the moveable sprites (now on higher-numbered planes) pass behind.

In text mode 1, sprites can be used as user-defined characters – you can augment the range of graphics characters built into the computer with sprite characters of your own design, and thus considerably widen the range of pictures which can be produced in this mode.

In graphics mode 2, static sprites can be used to eliminate the smudging effect which normally occurs if you attempt to use more than two different colours in any one block of eight pixels. The third and any subsequent colours can be added using sprites instead of the usual graphics commands.

Sprites can also be used to add fine details to displays produced in graphics mode 3. In this mode, pixels cannot be coloured individually, but only in 4×4 blocks; the screen resolution is comparatively low, but the sprites retain their usual high resolution. If only a few small areas of your planned screen displays require a high resolution, then the use of mode 3 with static sprites may be preferable to the use of mode 2.

14.6 SPRITE COLLISIONS

Collisions between two or more sprites can be detected by the computer

and used to trigger interrupts. This facility is very useful in arcade games, where the interrupt routine can be used to increase the player's score whenever a sprite 'bullet' connects with a sprite 'enemy', or to reduce the number of lives left whenever the hero is caught by a 'baddie'.

Once the interrupt has been set up it will be triggered when any two sprites overlap by a single pixel, even if that pixel is transparent rather than coloured. This can cause problems if, say, you are representing small bullets by much larger sprites with all but a few pixels set to transparent. The computer will register a hit even if the bullet has missed its target by several pixels.

No special means is provided of identifying the sprites which have collided. To do this, you must include in your interrupt routine a sequence of commands which will compare the X and Y co-ordinates of all the sprites on the screen. Remember that the sprite size and magnification will determine the range of difference between the co-ordinates which will produce an overlap.

The command used to specify the interrupt routine is:

ON SPRITE GOSUB <line no.>

The interrupt can be enabled/disabled/halted by:

SPRITE ON/OFF/STOP

As the sprites will continue to overlap while the interrupt routine is being executed, you must use SPRITE OFF (to disable further interrupts) at the start of the routine. If you do not do this, the routine will be repeated indefinitely. To re-enable the interrupt, use SPRITE ON again.

Example

This game involves moving a blue square around the screen with the cursor keys. The blue square is pursued by two red squares; if one of the red squares catches the blue square, or if the red squares collide, a life is lost. When five lives have been lost the score – based on the time the game lasted – is displayed.

```
10 REM Chase
20 SCREEN 1,0:WIDTH 30:COLOR 10,1,1:KEY OFF:CLS
30 SPRITE$(0)=STRING$(8,CHR$(255))
40 ON SPRITE GOSUB 500
50 PRINT STRING$(30,CHR$(219));
```

```
 60 FOR C=1 TO 21:PRINT CHR$(219)TAB(29)CHR$(219);:
    NEXT C
 70 PRINT STRING$(30,CHR$(219));
 80 SC=0:L=5
 90 X(0)=16:Y(0)=8:X(1)=232:Y(1)=166:X(2)=16:Y(2)=166
100 PUT SPRITE 0,(X(0),Y(0)),4,0
110 PUT SPRITE 1,(X(1),Y(1)),8,0
120 PUT SPRITE 2,(X(2),Y(2),8,0
130 SPRITE ON
140 A=STICK(0)
150 IF A>1 AND A<5 THEN IF X(0)<232
    THEN X(0)=X(0)+2
160 IF A>3 AND A<7 THEN IF Y(0) <166
    THEN Y(0)=Y(0)+2
170 IF A>5 THEN IF X(0)>16
    THEN X(0)=X(0)–2
180 IF A=8 OR A=1 OR A=2 THEN IF Y(0)>8
    THEN Y(0)=Y(0)–2
190 IF X(1)>X(0) THEN X(1)=X(1)–1 ELSE IF X(1)<X(0)
    THEN X(0)=X(0)+1
200 IF Y(1)>Y(C) THEN Y(1)=Y(1)–1 ELSE IF Y(1)<Y(0)
    THEN Y(1)=Y(1)+1
210 IF X(2)>X(0) THEN X(2)=X(2)–1 ELSE IF X(2)<X(0)
    THEN X(2)=X(2)+1
220 IF Y(2)>Y(0) THEN Y(2)=Y(2)–1 ELSE IF Y(2)<Y(0)
    THEN Y(2)=Y(2)+1
230 PUT SPRITE 0,(X(0),Y(0)),4,0
240 PUT SPRITE 1,(X(1),Y(1)),8,0
250 PUT SPRITE 2,(X(2),Y(2)),8,0
260 SC=SC+1
270 IF F=0 THEN GOTO 140
280 F=0
290 IF L>0 THEN GOTO 90
300 SCREEN 0:PRINT“SCORE =”;SC
310 END
500 SPRITE OFF
510 COLOR 10,8,8:FOR D=0 TO 50:NEXT D
520 COLOR 10,1,1
530 L=L–1:F=1
540 RETURN
```

Line 10: comment.

Line 20: selects text mode 1 with 8×8 unmagnified sprites, screen width 30 columns, foreground yellow, background and border black, removes the function key list from the bottom of the screen and clears the screen.
Line 30: defines a square sprite pattern.
Line 40: specifies the sprite collision interrupt routine (lines 500 on).
Lines 50, 60, 70: draw a yellow border around the screen.
Line 80: initializes the score (SC) and number of lives (L).
Line 90: sets the initial values of the sprite co-ordinates.
Lines 100–120: display sprites at initial co-ordinates.
Line 130: enables sprite collision interrupt.
Line 140: reads cursor keys.
Lines 150–180: update co-ordinates of blue square.
Lines 190–200: update co-ordinates of first red square.
Lines 210–220: update co-ordinates of second red square.
Lines 230–250: display sprites at new co-ordinates.
Line 260: increments score.
Line 270: returns to line 140 if flag (F) to indicate whether interrupt has been triggered is not set.
Line 280: resets flag.
Line 290: returns to line 90 if any lives remain.
Line 300: displays score.
Line 310: end.
Line 500 (start of interrupt routine): disables further interrupts.
Line 510: flashes background red.
Line 520: restores black background.
Line 530: decrements number of lives, sets flag.
Line 540: returns execution to main routine.

Exercise 14.1

What screen modes and sprite sizes will be selected by these statements?

(a) SCREEN 1,3
(b) SCREEN 3,0
(c) SCREEN 2,1

Exercise 14.2

Write a routine to define and display this multicoloured sprite, unmagnified, at co-ordinates (50,50) in graphics mode 2 with a light blue background:

```
---oo---
--oooo--        o:    medium red
-ooxxoo-
ooxxxxoo        x:    dark green
ooxxxxoo
-ooxxoo-        -:    transparent
---ooo---
--oooo--
```

Exercise 14.3

Write a routine which will display a Pac-Man-type character opening and closing its mouth. (Use two different patterns, one for the character with its mouth open and the other for the character with its mouth closed.)

15

Error Handling and Debugging

15.1 ERROR TRAPPING

When the BASIC interpreter detects an error during the execution of a program, it normally stops the execution and prints an error message to let you know what is wrong and in which program line the error occurred. There is a wide range of different error messages, which are listed in Appendix B. The computer uses two system variables when dealing with errors: **ERR**, to which it assigns the error code number, and **ERL**, to which it assigns the program line number.

Error-handling routines are similar to interrupt routines, except that errors cannot be enabled or disabled. The computer automatically takes some action when an error arises; you cannot prevent it from doing so, but you can change the form that the action takes. You cannot simply direct the computer to go back and carry on executing the program, as the error will prevent it from doing so; therefore the routine specifying command uses GOTO instead of GOSUB:

ON ERROR GOTO (line no.)

To print out the normal error message, use the command:

```
ON ERROR GOTO 0
```

Instead, you can use the values of ERR and ERL to print out your own message:

```
PRINT "Error code ";ERR;" in line no. ";ERL
```

Some errors can be dealt with within the error-trapping routine. For example, you could include a RESTORE command to solve an 'Out of

data' error (code 4). Others can be dealt with by simply skipping the statement in which they occur. The **RESUME** command can be used to return execution to the main program. This can take three forms:

RESUME or RESUME 0 resumes execution from the statement in which the error arose.
RESUME NEXT resumes execution from the statement after the one which caused the error.
RESUME (line no.) resumes execution from the specified line.

Example

```
10  ON ERROR GOTO 1000
20  FOR C%=0 TO 4
30  READ A(C%):PRINT A(C%):NEXT C%
40  DATA 0,1,2,3
50  LET B=3:PRONT B
60  FOR C%=1 TO 4
70  PRINT B/A(C%):NEXT C%
80  GOTO 20
1000  IF ERR=2 THEN PRINT "Syntax error in line ";ERL:RE-
      SUME NEXT
1010  IF ERR=4 THEN RESTORE:RESUME
1020  ON ERROR GOTO 0
```

Line 10: specifies error-handling routine.
Line 20: sets up loop to be executed for values of C% from 0 to 4.
Line 30: reads next data item, prints it on the screen; end of loop.
Line 40: data.
Line 50: sets value of B. The second statement in the line contains an error (PRONT instead of PRINT), and so cannot be executed.
Line 60: sets up loop to be executed for values of C% from 1 to 4.
Line 70: performs calculation, prints the result; end of loop.
Line 80: returns to line 20.
Line 1000 (start of error-handling routine): if error is 'Syntax error', prints message then returns execution to the following line.
Line 1010: if error is 'Out of data', restores data then returns to the READ statement where the error arose.
Line 1020: prints the usual error message.

RUN this, and you should get:

```
 0
 1
```

```
 2
 3
 0
Syntax error in 50
 3
 1.5
 1
Division by zero in 70
Ok
```

The program contained three errors. The first was an 'Out of data' error (code 4); the data was restored, so A(4) was given the value 0. The second was a 'Syntax error' (code 2); a special error message was printed, and the computer ignored the erroneous statement (PRONT B) and resumed execution from the next statement. The third error was a 'Division by zero' error; this time the normal error message was printed and the program execution was halted.

You can create your own errors if you wish, using an **ERROR** statement. Code numbers 61 to 255 can be used for your own errors, which can then be dealt with by an error-trapping routine. The statement to create an error takes the form:

```
IF (condition) THEN ERROR (error code)
```

Example

```
 10 ON ERROR GOTO 100
 20 INPUT "Give me a positive number";A
 30 IF A<0 THEN ERROR 255
 40 PRINT A
 50 GOTO 20
100 IF ERR=255 THEN PRINT "I said I wanted a positive
    number! Try again.":RESUME 20
110 ON ERROR GOTO 0
```

Line 10: specifies error-handling routine.
Line 20: inputs a number and assigns it to the variable A.
Line 30: creates an error.
Line 40: prints the value of A.
Line 50: returns to line 20.
Line 100: (start of error-handling routine): if error is the error which was created in line 30, prints message then returns execution to line 20.
Line 110: prints the usual error message.

Exercise 15.1

A circle centre (a,b), radius r can be drawn by plotting the points (a+r*sin(*t*),*b*+*r**cos(*t*)) for values of *t* from 0 in increments of 0.01 to 2*PI. (2#). Write statements to create an error if the co-ordinate variables X and Y are outside the ranges 50<=X=206, 18<=Y<=174, and an error-trapping routine which will deal with this error by setting X to 50 if X<50 etc. Incorporate these into a program to plot a circle centre (128,96), radius 80, to produce a program which will draw a circle with flattened edges.

15.2 PROGRAM BUGS

If you have written a program which is more than just a few lines long, the chances are that it will not run properly at the first attempt. You may have made mistakes when writing the program, or typing errors when entering it, or both. The program may be halted by an error message, or it may do something which you did not expect. It can take longer to track down and eliminate the *bugs* which prevent the program from working properly than it took to write it! However, there are several commands and techniques which can be used to help with the debugging process.

It is always easier to find mistakes in a well-structured program than in one which is full of unnecessary GOTOs. If your program is made up of clearly defined routines, linked together in a logical order, then you should be able to tell quite easily which sections are causing problems. You may even be able to debug the routines individually, instead of having to tackle the whole program at once.

A list of the variables and their functions is very useful. In a long program you could cause problems by using the same name for two different variables, an error which is easy to spot if you have drawn up a variable list.

Example

```
10 SCREEN 2:COLOR 1,15,15:CLS
20 PI=4*ATN(1)
30 PSET(10,90)
40 FOR X=0 TO 2*PI STEP 0.1
50 Y=SIN(X)
60 Y=90+50*Y
70 X=10+X*245/(2*PI)
```

```
 80 LINE -(X,Y)
 90 NEXT X
100 A$=INPUT$(1)
110 END
```

Line 10: selects graphics mode 2, foreground black, background and border white, and clears the screen.
Line 20: defines the value of PI (π).
Line 30: moves the graphics cursor to (10,90) (and sets the point (10,90) to black).
Line 40: sets up loop to be executed for values of X from 0 in increments of 0.1 to 2*PI.
Line 50: calculates the sine of X, and assigns it to the variable Y.
Line 60: scales the value of Y to produce a Y co-ordinate between 40 and 140.
Line 70: scales the value of X to produce an X co-ordinate between 10 and 10+60*PI.
Line 80: draws a black line to the point (X,Y).
Line 90: end of loop.
Line 100: holds the display on the screen until a key is pressed.
Line 110: end of program.

This program should draw a sine wave, but if you RUN it you will see nothing but a black dot on the screen. The bug is quite easy to spot: the variable X has been used as the loop counter, and as a screen co-ordinate. Substituting SX for the first X in line 70 and the X in line 80 will solve the problem:

```
70 SX=10+X*245/(2*PI).
80 LINE-(SX,Y)
```

Now when you RUN the program the sine wave will be drawn.

Typing errors normally produce syntax errors, which can be found and corrected quite easily. When the computer prints out an error message, you can list the line in which the error arose by pressing <F9>. Check the line which is displayed with your written program listing, and the error should be obvious.

If you make a typing error in a DATA statement, the error message which is produced (if there is one) will almost certainly refer to the line in which the data is read, not to the data statement itself. If, then, an error arises in a line containing a READ instruction, you should look

carefully at the corresponding DATA statement. If your program contains a large amount of numeric data, it is a good idea to include a *checksum* routine, so that you can find any mistakes in the data easily. At the end of each batch of numbers, put in an extra data item giving their total. When the data is read, use a special checksum variable to keep a running total, then compare the value of this variable with the total given in the DATA statement.

Example

```
10 DIM X(20)
20 S=0
30 FOR C=1 TO 20
40 READ X(C):S=S+X(C)
50 NEXT C
60 READ T
70 IF T<>S THEN PRINT"Mistake in data":END
80 DATA 5,3,7,9,4,2,6,8,5,6,9,4,6,8,2,4,3,3,9,7,110
```

Line 10: dimensions array.
Line 20: initializes checksum variable.
Line 30: sets up loop, to be executed for values of C from 1 to 20.
Line 40: reads next data item, assigns it to X(C) and adds it to the checksum variable.
Line 50: end of loop.
Line 60: reads data total.
Line 70: compares total with checksum variable, prints error message if necessary.
Line 80: data (twenty items), data total.

This routine reads twenty numbers into an array, and provides a check on the accuracy of the data.

Mistakes in graphics routines are rarely difficult to find, as their results are visible. It can be helpful to put in a temporary delay loop, or an INPUT$ statement to freeze the picture until a key is pressed, so that you can see exactly what is happening. If part of the screen display does not appear, it will probably be because you have accidentally drawn it in the background colour! Mistakes in sound routines are also usually obvious.

When calculations have gone haywire, inserting statements to print out the values of the variables involved before and after the calculation is performed can often help you to discover where the fault lies. The

values of variables can also be printed out in direct mode after execution of the program has been stopped.

Many bugs arise because program lines have been placed in the wrong order. It is all too easy to accidentally undimension an array by putting a CLEAR command after the DIM statement, or to forget to use a SCREEN command to set the sprite size before defining a sprite pattern. You cannot use RENUM to change the order of program lines, but it is possible to move a line without typing it in again: LIST the line you want to move, then type the new line number over the old one and press <RETURN>. You will now have two lines which are identical apart from their numbers; simply delete the original line (by typing the line number <RETURN>, and your line will have been moved. (This technique is also useful if you have several identical lines to enter.)

Among the most difficult bugs to find are those which affect the order of execution of a program. To help you to deal with these, there is a special *trace* facility which makes the computer display the line number of each line as it is executed. To switch this facility on, use the command **TRON**. The list of line numbers should enable you to see easily if there is a jump to the wrong place, or if a loop is not being executed the correct number of times. The command to switch the trace off again is **TROFF.**

Example

```
10 REM Days in the month
20 INPUT "Which month (1–12)";M
30 ON M GOTO 50,60,50,70,50,70,50,50,70,50,70,50
40 PRINT"Invalid month no.":GOTO 20
50 N=31
60 N=28
70 N=30
80 PRINT"Month";M;" has";N;" days"
90 GOTO 20
```

Line 10: comment.
Line 20: prints prompt, inputs a number and assigns it to M.
Line 30: jumps to line 50 if M = 1, 3, 5, 7, 8, 10 or 12, line 60 is M = 2, line 70 if M = 4, 6, 9 or 11.
Line 40: prints message, returns to line 20.
Line 50: sets N (number of days) to 31.
Line 60: sets N to 28.
Line 70: sets N to 30.
Line 80: prints and month and number of days.
Line 90: returns to line 20.

If you RUN this program, you will find that it tells you every month has 30 days. Enter TRON, and you will get:

```
[10][20]Which month (1–12)? 1
[30][50][60][70][80]Month 1 has 30 days
[90][20]Which month?
```

The program execution started with line 10, then line 20. After the input of 1, line 30 sent it to line 50 – so far, so good – but then it went on to lines 60 and 70, instead of going straight to line 80! There are GOTOs missing from the end of lines 50 and 60. Edit the program to include them:

```
50  N=31:GOTO 80
60  N=28:GOTO 80
```

Now it should work properly.

One limitation of the trace facility is that it does not work in the graphics modes. The line numbers can only be displayed on the text screen. If you want to know whether a particular statement is being executed in a graphics program, one good way to find out is to follow it with a BEEP command. If there is no beep when you run the program, you will know that the command is being bypassed somehow.

No exercise is provided here – you should have plenty of opportunities to practise debugging techniques while writing your own programs!

16

Introduction to Machine Code

16.1 MACHINE CODE AND ASSEMBLY LANGUAGE

The Z80 processor, which is at the heart of an MSX computer, does not understand BASIC. Its 'language' is *machine code*: a set of instructions known as *opcodes* which are represented, not by easily understood keywords, but by hexadecimal numbers. This book will not teach you to write machine-code programs, only how to use existing machine-code routines within BASIC programs. If you want to learn more about Z80 machine-code, there are a number of good specialist books available to help you.

The processor contains some special memory locations called *registers*, which are labelled A, B, C, D, E, H, L and F. The A register is the *accumulator*, and can be used for performing simple arithmetic. The F register is the *flag* register; its eight bits indicate the outcome of various operations. The other registers can be used singly or in pairs to store 8 bit or 16 bit numbers.

The machine-code instruction set includes instructions to load numbers into the registers, to store the contents of the registers in specified memory locations, to increment or decrement the contents of the registers, to perform simple addition and subtraction, to perform Boolean logic operations (AND, OR etc.), and to rotate the bits in a register to the right or the left. There are also instructions which enable you to set up program branches, depending on the value of one of the bits of the F register, and to use subroutines.

Each machine-code instruction occupies between 1 and 4 bytes of memory. A machine-code program consists of a long string of two-digit hexadecimal numbers, which have to be stored in successive memory locations within the computer's RAM.

A program which is composed entirely of hexadecimal numbers is

difficult to write, and almost impossible to understand, so most machine-code programs are written using mnemonics to represent the instructions, in place of the hex opcodes. These mnemonics form a language called *assembly language*. A program written in assembly language is not as easy to follow as one written in BASIC, but it is a lot more comprehensible than machine code. The Z80 assembly language instructions and opcodes are given in Appendix H.

A special program called an *assembler* is to translate the assembly language program into machine code. A *disassembler* performs the reverse operation: it translates machine code into assembly language. The BASIC editor cannot be used to edit machine-code programs, so a special program has to be used for this too: a *monitor*. A good monitor can make the process of entering and debugging machine-code programs infinitely easier. If you intend to write your own machine-code programs, you will need an assembler and a monitor. A disassembler is useful but not essential.

The BASIC interpreter and operating system are written in machine code, and contain a lot of useful routines (BASIC Input/Output Systems or BIOS routines) which you can use in your own machine-code programs. Any good book on MSX machine-code programming should explain what these are and how to use them.

16.2 ENTERING MACHINE-CODE ROUTINES

If you want to use a machine-code subroutine with a BASIC program, you must clear space in RAM for the machine code to ensure that it will not occupy the same space as the BASIC program, variables or work area. This can be done using the command **CLEAR**, which is also used to reserve space for string storage (see Section 6.2). The syntax is:

CLEAR <string space>,<highest memory location available to BASIC>

The way in which the available memory is allocated to various functions is shown by the memory map in Appendix G. When the computer is first switched on, the file control block ends at location &HF380. If you enter a lower location number as the second parameter in the CLEAR command, the stack area, string storage area and file control block will be moved down so that the file control block ends at the location specified, and the area between this location and &HF380

is available for machine code. For example, to clear space for a machine code routine 50 bytes long, use:

```
CLEAR 255,&HF330
```

Once you have cleared space for the machine code, the next task is to put the numbers making up the routine into the appropriate memory locations. To do this, use the BASIC command **POKE**:

```
POKE<memory location>,<number>
```

The number which you POKE into a memory location must be only 1 byte long, i.e. 8 binary digits, 2 hex digits, or a decimal integer between 0 and 255.

A short BASIC loader routine can make the entry of a machine-code routine simple:

```
10  REM Machine-code loader
20  INPUT"Start location";S
30  INPUT "How many bytes";N
40  FOR L=S TO S+N
50  PRINT L;":       ";:INPUT H$
60  POKE L,VAL("&H"+H$)
70  NEXT L
80  END
```

Line 10: comment.
Line 20: prints prompt, inputs start location for machine code.
Line 30: prints prompt, inputs number of bytes to be entered.
Line 40: sets up loop, to be executed for values of L from S (start location) to S+N (end location).
Line 50: prints location, inputs byte (in hex).
Line 60: pokes byte into location.
Line 70: end of loop.
Line 80: end of program.

If your data is in decimal, not hex, just change lines 50 and 60:

```
50  PRINT L;":         ";:INPUT D
60  POKE L,D
```

If the machine-code routine is to be used as a subroutine of a BASIC program, you can instead put the machine-code data into DATA statements within the BASIC program and include a similar loader routine at the start of the program, so the machine code is loaded

automatically when you run the BASIC program.

The reverse statement to POKE is **PEEK**, which returns the contents of a memory location:

PEEK(<memory location>)

You can use PEEK to examine the contents of the BASIC ROM if you wish (locations &H0000 to &H8000 for a 64K MSX computer).

There are special variants of PEEK and POKE for use with the video RAM – see Chapter 17.

16.3 EXECUTING MACHINE-CODE ROUTINES, PASSING AND RETURNING PARAMETERS

Once you have entered your machine-code routines, the next task is to tell the computer where they start, so they can be called from a BASIC program. The command used is **DEF USR**:

DEF USR[<routine no.>]=<start location>

The routine number must be an integer between 0 and 9 (so you can use ten different routines at once!). If the number if omitted, the default is 0.

The routines which you have entered and defined can be called by using the **USR** function. This function is also used to pass parameters from the BASIC program to the machine-code routine, and back again. Its format is:

USR<routine no.>(<parameter>)

The parameter cannot be omitted, so if you do not want to pass a parameter to the machine-code routine, just put a 0 inside the brackets.

When a USR function is used, the BASIC interpreter sets location &HF663 to a value which indicates the type of the parameter. The values assigned to this location are:

2	integer
3	string
4	single-precision number
8	double-precision number

The memory locations in which the parameter itself is stored depend on its type:

Integer Low byte &HF7F8, high byte &HF7F9.
Single-precision number Stored in binary-coded decimal format (four binary digits are used to represent each decimal digit) in locations &HF7F6 to &HF7F9.
Double-precision number Stored in binary-coded decimal in locations &HF7F6 to &HF7FD.
String Locations &HF7F8 and &HF7F9 give the low and high bytes of the memory location which contains the string length. The following locations contain the characters of the string.

When execution is returned from the machine-code routine to the BASIC program, the contents of these locations determine the value of the USR function.

Examples

```
10 CLEAR 255,&HF370
20 DEF USR=&HF370
30 DEF USR1=&HF372
40 DEF USR2=&HF374
50 DEF USR3=&HF376
60 FOR C=&HF370 TO &HF376 STEP 2:POKE C,&HC9:
   NEXT C
70 X%=USR(5)
80 PRINT X%
90 Y!=USR1(4.35!)
100 PRINT Y!
110 Z#=USR2(12.63#)
120 PRINT Z#
130 A$=USR3("ABC")
140 PRINT A$
150 END
```

Line 10: clears 255 bytes of string storage, reserves locations &HF370 to &HF380 for machine-code routines.
Lines 20–50: specify start of routines 0, 1, 2 and 3 as &HF370, &HF372, &H374 and &H376 respectively.
Line 60: pokes the opcode &HC9 (return from subroutine) into start locations of routines 0, 1, 2 and 3.
Lines 70–140: call each routine in turn, passing to them the parameters

5, 4.5!, 12.63# and "ABC" respectively, and print the parameters returned.
Line 150: end of program.

This program defines four machine-code subroutines, each of which consists just of the instruction 'RETurn', and calls them in turn, passing a different type of parameter to each of them. As the routines do nothing, the parameters returned are the same as the parameters passed, so if you RUN the program you will get:

```
5
4.5
12.63
ABC
```

Example

```
10 CLEAR 255,&HF300
20 FOR C=&HF300 TO &HF300+16: READ A$:POKE C,VAL
   ("&H"+A$):NEXT C
30 DATA 3E,06,26,F7,2E,F9,77,3D,2B, 77,2B,3E,02,32,F6,63,C9
40 DEF USR=&HF300
50 A=USR(54)
60 PRINT A
70 END
```

Line 10: clears 255 bytes for string storage, reserves locations &HF300 to &HF380 for machine-code routines.
Line 20: reads machine-code data and pokes it into locations &HF300 to &HF316.
Line 30: machine-code data.
Line 40: specifies start of routine 0 as &HF300.
Line 50: calls routine 0, passing parameter 54 to the routine, and assigns the parameter returned to the variable A.
Line 60: prints A.
Line 70: end of program.

This is the machine-code routine:

```
3E06    LD A,&H06     puts the number 6 into register A
26F7    LD H,&HF7     puts &HF7 into register H
2EF9    LD L,&HF9     puts &HF9 into register L
```

```
77        LD (HL),A      puts the number in A into &HF7F9
3D        DEC A          subtracts 1 from the number in A
2B        DEC HL         subtracts 1 from the two-byte number in HL
77        LD (HL),A      puts the number in A into &HF7F8
2B        DEC HL         subtracts 1 from the two-byte number in HL
3E02      LD A,2         puts the number 2 into register A
32F663    LD &HF663,A    puts the number in A into &HF663
C9        RET            returns to BASIC
```

At the end of this routine, register A and location &HF663 both contain the number 2, indicating that the parameter returned from the routine is an integer. The high byte of this integer is given by location &HF7F9, which contains 6. The low byte is given by location &HF7F8, which contains 5. The register pair HL contains the address &HF7F6. The parameter returned should therefore be 256*6+5, or 1541. RUN the BASIC program to check that it does print this number.

Exercise 16.1

Write a BASIC program which will load, then execute, this very short machine-code routine:

```
CDC000    CALL &H00C0
C9        RET
```

(The program should simply produce a beep.)

17

The Video RAM and Video Processor

17.1 THE VIDEO RAM

MSX computers have a special section of memory, the *video RAM (VRAM)*, which is dedicated to the video display. There is 16K (16384 bytes) of this memory. The way in which it is used depends on the screen mode, and the contents of the VDP registers (see Section 17.5). The position of various screen tables within the VRAM can be determined using the **BASE** function. The argument of BASE must be an integer between 0 and 19, and the function returns the following addresses:

BASE(0)	start of pattern name table in mode 0
BASE(1)	unused
BASE(2)	start of pattern generator in mode 0
BASE(3)	unused
BASE(4)	unused
BASE(5)	start of pattern name table in mode 1
BASE(6)	start of colour table in mode 1
BASE(7)	start of pattern generator in mode 1
BASE(8)	start of sprite attribute table in mode 1
BASE(9)	start of sprite pattern table in mode 1
BASE(10)	start of pattern name table in mode 2
BASE(11)	start of colour table in mode 2
BASE(12)	start of pattern generator in mode 2
BASE(13)	start of sprite attribute table in mode 2
BASE(14)	start of sprite pattern table in mode 2
BASE(15)	start of pattern name table in mode 3
BASE(16)	unused
BASE(17)	start of pattern generator table in mode 3
BASE(18)	start of sprite attribute table in mode 3
BASE(19)	start of sprite pattern table in mode 3

Values can be assigned to memory locations within the VRAM using the command **VPOKE** (the video RAM equivalent of the command POKE, used to assign values to memory locations in the main memory: see Chapter 16). The contents of memory locations can be read by using the function **VPEEK** (the VRAM version of PEEK).

17.2 THE PATTERN NAME TABLE

Modes 0 and 1

In the two text modes (mode 0 and mode 1), the pattern name table contains the ASCII codes for the characters displayed in each position on the screen. Mode 0 has 40 columns and 24 rows of characters, making a total of 40×24 = 960 positions. Each of these positions is allocated one byte of the video memory. BASE(0) returns the start position of the map; the memory address containing the character displayed at (X,Y) can be calculated from the formula:

address of (X,Y) in mode 0 = BASE(0)+X+40*Y

Mode 1 has 32 columns and 24 rows, so its pattern name table occupies 32×24 = 768 bytes of memory. The formula to be used here is:

address of (X,Y) in mode 1 = BASE(5)+X+40*Y

This information can be used to VPOKE characters on to the screen.

Example

```
10 SCREEN 1:WIDTH 32:COLOR 15,4,4:CLS
20 S=BASE(5)
30 FOR C=0 TO 255
40 VPOKE S+C,C
50 NEXT C
60 GOTO 60
```

Line 10: selects text mode 1, screen width 32 columns, foreground white, background and border blue, and clears the screen.
Line 20: sets S to the start address of the pattern name table.
Line 30: sets up loop to be executed for values of C from 0 to 255.

Line 40: pokes character C into next address in pattern name table.
Line 50: end of loop.
Line 60: holds display on the screen.

This program will display all the available characters.

Example

```
10 SCREEN 0:WIDTH 36:COLOR 1,7: CLS
20 LOCATE 0,10:PRINT“scroll”
30 S=BASE(0)
40 FOR N=2 TO 37
50 L=37:IF N+6<37 THEN L=N+6
60 FOR C=L TO N STEP -1
70 VPOKE S+400+C,VPEEK(S+399+C)
80 NEXT C,N
90 END
```

Line 10: selects text mode 0, screen width 36 columns, foreground black, background sky blue, and clears the screen.
Line 20: moves the text cursor to column 0, row 10, and prints ‘scroll’.
Line 30: sets S to the start address of the pattern name table.
Line 40: sets up a loop to be executed for values of N from 2 to 37.
Line 50: sets L to 37 or N+6, whichever is the lower.
Line 60: sets up a loop to be executed for values of C from L in increments of –1 to N (a maximum of six times, once for each letter to be scrolled).
Line 70: moves the word ‘scroll’ one column to the right by poking each letter code into the next location (starting from the right-hand letter).
Line 80: end of loops.
Line 90: end of program.

This program scrolls the word ‘scroll’ across the screen. Scrolling is a slow process in BASIC; only the six letters of the word are scrolled, because it would take too long to scroll the whole screen. (You would have to use a machine-code routine to do that.) The start value of inner loop is set to a maximum of 37 so that the word will disappear off the right-hand side of the screen, and not reappear at the left.

Mode 2

The mode 2 (high-resolution graphics) screen consists of 32 columns

and 24 rows of character positions. Each character position is allocated 1 byte of memory in the pattern name table, as for the text screens. The 768 bytes in the table are initially set to the numbers 0–255, 0–255, 0–255. These numbers refer to three blocks of 256 character patterns in the pattern generator table (see Section 17.3). Each pattern in the pattern generator table is displayed once, to make up the picture produced on the screen using the normal graphics commands. The screen is divided horizontally into thirds; the first block of 256 character patterns makes up the top third of the picture, the second block of patterns makes up the middle third of the picture, and the third block of patterns makes up the bottom third. By VPOKEing the pattern name table with an appropriate value, you can replace the contents of any character position with the contents of any other character position in the same third of the screen. This provides you with an easy way of duplicating a drawing on a different part of the screen.

Example

```
10 SCREEN 2:COLOR 2,15,15:CLS
20 S=BASE(10)
30 DRAW"BM1,136;E4F4L8":PAINT(4,134)
40 CIRCLE(3,68),3,5:PAINT(3,68),5,5
50 LINE(2,2)-(6,6),8,BF
60 FOR C=S TO S+767:VPOKE C,0:NEXT C
70 GOTO 70
```

Line 10: selects graphics mode 2, foreground green, background and border white, and clears the screen.
Line 20: sets S to the start address of the pattern name table.
Line 30: draws a green triangle in the first character position in the bottom third of the screen.
Line 40: draws a blue circle in the first character position in the middle third of the screen.
Line 50: draws a red square in the first character position in the top third of the screen.
Line 60: VPOKEs a 0 into every memory location in the pattern name table.
Line 70: Holds the display on the screen.

When you run this program, you will see a red square, a blue circle and a green triangle drawn at equal intervals down the left-hand side of the screen. Then the top, middle and bottom thirds of the screen will be filled with red squares, blue circles and green triangles respectively.

Mode 3

The low-resolution mode 3 screen also has 32 columns and 24 rows of character positions, and the pattern name table again occupies 768 bytes of memory, 1 byte for each character position. They initially contain the values 0–31, 0–31, 0–31, 0–31, 32–63, 32–63, 32–63, 32–63, 64–95,. . ., 160–191. These numbers give a clue to the arrangement of the pattern generator table: this contains four blocks of 192 character patterns. The first block contains the patterns for screen rows 0, 4, 8, 12, 16 and 20, the second block contains the patterns for rows 1, 5, 9, 13, 17 and 21, etc. The pattern in any character position can be replaced by another character pattern in the same block, by VPOKEing a number between 0 and 191 into the appropriate location in the pattern name table. VPOKEing a number between 192 and 255 will blank the character position.

Example

```
10 SCREEN 3:COLOR 1,15,15:CLS
20 S=BASE(15)
30 PSET(0,0),2:PSET(0,8),4:PSET(0,16),6:PSET(0,24),10
40 FOR C=S TO S+767:VPOKE C,0:NEXT C
50 GOTO 50
```

Line 10: selects graphics mode 3, foreground black, background and border white, and clears the screen.
Line 20: sets S to the start address of the pattern name table.
Line 30: puts small green, blue, red and yellow squares in the first character positions in rows 0, 1, 2 and 3 respectively.
Line 40: VPOKEs a 0 into every memory location in the pattern name table.
Line 50: Holds the display on the screen.

This program will produce alternate rows of green, blue, red and yellow squares.

17.3 THE PATTERN GENERATOR AND COLOUR TABLES

Modes 0 and 1

In the text modes, the pattern generator table contains the patterns for all the 256 standard characters. Each pattern occupies 8 bytes of memory, 1 byte for each row of eight pixels. The start address for character *n* can be determined from the formulae:

start address in mode 0 = BASE(2)+8**n*
start address in mode 1 = BASE(7)+8**n*

It is possible to define your own characters to replace one or more of the standard characters. Characters are defined in the same way as 8×8 sprites (see Chapter 14); the eight numbers defining the pattern are VPOKEd into the eight consecutive memory locations in the pattern generator table which correspond to the character which is being replaced.

Example
To replace character 65 (A) with an arrow.
First design the character and calculate the data:

--------	0
----*---	8
-----*--	4
------*-	2
********	255
------*-	2
-----*--	4
----*---	8

Then write a program to VPOKE the data into the pattern generator table:

```
10 SCREEN 1
20 S=BASE(7)
30 FOR C=0 TO 7
40 READ X:VPOKE S+65*8+C,X
50 NEXT C
60 DATA 0,8,4,2,255,2,4,8
```

Nothing will appear to happen when you RUN this program, but if you LIST it, you will see that the As in 'BASE' and 'DATA' have been replaced by arrows!

The standard characters can be restored by using another SCREEN command.

Remember that in mode 0 the character positions are only six pixels wide, so when you design characters for use in this mode, leave the two rightmost columns of pixels in your 8×8 grid blank.

There is no colour table for mode 0, as only two colours can be used on the screen at any one time in this mode.

The colour table for mode 1 is 32 bytes long. Each byte determines the foreground and background colours for a block of eight character patterns; the first byte in the table determines the colours of characters 0 to 7, the second byte determines the colours of characters 8 to 15, and so on. Bits 7 to 4 determine the foreground colour and bits 3 to 0 determine the background colour. So, for example, to select foreground white (colour code 15, or &B1111) and background dark blue (colour code 4, or &B0100), you must VPOKE the number &B11110100, decimal 244, into the colour table (note that the number to be VPOKED = 16*< foreground colour code> + <background colour code>).

The location corresponding to character code *n* (and the seven other characters in the same block) can be determined from the formula:

address for character *n* in colour table, mode 1 = BASE(6)+*n*\8

Example

```
10 SCREEN 1
20 FOR C=0 TO 15:VPOKE BASE(6)+C,C:NEXT C
30 END
```

Line 10: selects text mode 1.
Line 20: VPOKES the numbers 0 to 15 into the first fifteen locations in the colour table.
Line 30: end of program.

When you run this program, the background colours of the characters on the screen will change. The program listing will look very odd!

Mode 2

The pattern generator table in mode 2 contains 768 character patterns

– one for each character position on the screen. The patterns are coded and stored in the same way as the standard character patterns – 8 bytes of memory are used to represent the contents of each character position. The characters in the top row of the screen are stored first, starting with the leftmost, then the characters for the second row, and so on.

The fact that the pattern generator table does not contain the standard character patterns explains why the normal PRINT commands cannot be used in this mode. In order to print characters on the mode 2 screen, the character patterns stored in ROM must be transferred to the VRAM through a file channel.

The contents of each memory location in the colour table for this mode determine the foreground and background colours for one row of eight pixels in one character position. Bits 7 to 4 determine the foreground colour of the row of pixels, and bits 3 to 0 determine the background colour (as in the colour table for mode 1).

It is possible to produce pictures by VPOKEing numbers into the pattern generator and colour tables; however, it is difficult to imagine why anyone would want to do it this way in a BASIC program, when the same results can be achieved much more easily by using the graphics commands (LINE, CIRCLE, DRAW etc.) The addresses in the colour and pattern generator tables corresponding to a point (X,Y) can be calculated from this formula:

address = BASE(*n*)+32*8*(Y \8)+8*(X \8)+YMOD8

where *n*=11 for the colour table, 12 for the pattern generator table.

As each address refers to a row of eight pixels, to set the colour of a particular pixel you also need to know which bit of the byte in the pattern generator table determines its status. This can be calculated using the formula:

bit = 2^(7–XMOD8)

The Boolean operators AND and OR can be used to change the value of a single bit in a byte, without altering the value of the other seven bits. For example, to set bit 3 of memory location A to 1:

```
VPOKE A,(VPEEK(A) OR &B00001000)
```

To set bit 5 of location A to 0:

```
VPOKE A,(VPEEK(A) AND &B11011111)
```

Example

```
10 SCREEN 2:COLOR 15,4,4:CLS
20 FOR C=BASE(11) TO BASE(11)+&H17FF:POKE
   C,243:NEXT C
30 FOR X=0 TO 100
40 A=BASE(12)+32*8*(X\8)+XMOD8
50 VPOKE A,(VPEEK(A) OR 2^(7-XMOD8))
60 NEXT X
70 GOTO 70
```

Line 10: selects graphics mode 2, foreground white, background and border dark blue, and clears the screen.
Line 20: pokes 243 (foreground white, background light green) into every location in the colour table.
Line 30: sets up a loop to be executed for values of X from 0 to 100.
Line 40: calculates the address in the pattern generator table corresponding to the point (X,X).
Line 50: sets the bit corresponding to the point (X,X) to the foreground colour.
Line 60: end of loop.
Line 70: holds the display on the screen.

When you RUN this program you will see the dark blue screen change gradually to light green, then a white line will be drawn from (0,0) to (100,100).

Mode 3

There is no colour table in mode 3, because the pattern generator table doubles as a colour table. Two bytes in this table are allocated to each character position. The character positions are divided into four 4×4 pixels squares; the first byte contains the colour codes for the top left square (bits 7–4) and the top right square (bits 3–0), and the second byte contains the colour codes for the bottom left square (bits 7–4) and the bottom right square (bits 3–0).

The address corresponding to a point with screen co-ordinates (X,Y) can be calculated from the formula:

```
address = BASE(17)+32*8*(Y\32)+8*X\8+(YMOD32)\4
```

If XMOD8<4, the colour code of the point (X,Y) will be given by bits 7–4; if XMOD4>=4, the colour code will be given by bits 3–0.

Example

```
10 SCREEN 3:COLOR 1,15,15:CLS
20  PI=4*ATN(1)
30  FOR X%=0 TO 255 STEP 4
40  Y%=96+90*SIN(X%*2*PI/255)
50 A=BASE(17)+32*8*(Y%\32)+8*(X%\B)+(Y%MOD32)\4
60 IF  X%MOD4<4  THEN  VPOKE  A,(VPEEK(A)  AND
    &B00111111) ELSE VPOKE A,(VPEEK(A) AND &B11110011)
70 NEXT X%
80 GOTO 70
```

Line 10: selects graphics mode 3, foreground black, background and border white, and clears the screen.
Line 20: defines the value of PI.
Line 30: sets up a loop to be executed for values of X% from 0 in increments of 4 to 255.
Line 40: calculates the value of Y%
Line 50: assigns the address in the pattern generator table corresponding to the point (X%,Y%) to A.
Line 60: VPOKEs the pattern generator table, to set the 4×4 pixel square containing the point (X%,Y%) to light green (colour code 3, or &B0011).
Line 70: end of loop.
Line 80: holds the display on the screen.

This program plots a sine wave, scaled to fit nicely on the screen.

17.4 THE SPRITE ATTRIBUTE AND SPRITE PATTERN TABLES

These two tables are arranged in the same way in all three modes which permit the use of sprites (modes 1, 2 and 3). Each of the 256 available 8×8 sprite patterns occupies 8 bytes of memory in the sprite pattern table; these patterns are numbered from 0 to 255, and the start address for sprite pattern *n* is:

BASE(5*<screen mode>+4)+8**n*

Size 16×16 sprites are treated as four 8×8 sprites stuck together. The

8×8 sprite patterns are numbered individually even if you are using 16×16 sprites, so 16×16 sprite pattern *n* consists of 8×8 sprite patterns *n**4, *n**4+1, *n**4+2 and *n**4+3.

Sprite patterns can be defined by VPOKEing the pattern data into the appropriate locations, instead of using the SPRITE$ command.

Example

To define sprite pattern 3 in mode 2 with 8×8 unmagnified sprites.

pattern	*data*
--------	0
---**---	24
--****--	60
--****--	60
--*--*--	36
--*--*--	36
-*----*-	66
------	129

```
10 MODE 2,0
20 FOR C=BASE(14)+24TO BASE(14)+31
30 READ X:VPOKE C,X:NEXT X
40 DATA 0,24,60,60,36,36,66,129
```

Line 10: selects graphics mode 2 with 8×8 unmagnified sprites.
Line 20: sets up loop to be executed eight times.
Line 30: reads sprite data and VPOKES it into memory locations for sprite pattern 3.
Line 40: Sprite data.

Each of the 32 available sprites is allocated 4 bytes of memory in the sprite attribute table. These bytes determine the Y co-ordinate and X co-ordinate of the top left-hand corner of the sprite, the sprite pattern number and the colour, respectively. The start address for the data for sprite number *n* is:

BASE(5*<screen mode>+3)+4**n*

In the case of 16×16 sprites, the sprite pattern number may be the number of any one of the four constituent 8×8 sprites, i.e. 16×16 sprite pattern number *n* can be selected by VPOKEing 4**n*, 4**n*+1, 4**n*+2, or 4**n*+3 into the appropriate address.

Example

```
10 SCREEN 1,2:COLOR 1,15,15:CLS
20 FOR C=BASE(9) TO BASE(9)+31:VPOKE C,255:VPOKE
   C+32,170:NEXT C
30 S=BASE(8)
40 VPOKE S,100:VPOKE S+1,50:VPOKE S+2,2:VPOKE S+3,8
50 VPOKE S+4,100:VPOKE S+5,100:VPOKE S+6,7:VPOKE
   S+7,4
60 GOTO 60
```

Line 10: selects text mode 1 with 16×16 unmagnified sprites, foreground black, background white, and clears the screen.
Line 20: defines sprite patterns 0 to 3 as solid squares, sprite patterns 4 to 7 as vertically striped squares.
Line 30: sets S to the start address of the sprite attribute table.
Line 40: sets the data for sprite number 0: Y co-ordinate 100, X co-ordinate 50, sprite pattern number 2, colour 8 (medium red).
Line 50: sets the data for sprite number 1: Y co-ordinate 100, X co-ordinate 100, sprite pattern number 7, colour 4 (dark blue).
Line 60: holds the display on the screen.

When you RUN this program, you should see a solid red 16×16 sprite at (50,100) and a striped blue and white 16×16 sprite at (100,100).

17.5 THE VIDEO PROCESSOR REGISTERS

The processor which handles the video display has nine registers, the contents of which determine how the video RAM is used. The first eight registers (registers 0 to 7) are write only, so you can assign values to them but you cannot directly examine their contents. The ninth register (register 8) is read only.

The function **VDP** is used to access the registers. To assign a value to one of the write-only registers, use the format:

VDP(<register no.>)=<value>

and to read the contents of register 8, use:

<numeric variable>=VDP(8)

If you assign a wrong number to one of the registers, the display may break up and you will have to switch the computer off and on again to restore the original register contents, so use the VDP function with caution.

The registers each contain 8 bits (binary digits), numbered 7 (the leftmost bit) to 0 (the rightmost bit). Their functions are as follows.

Register 0

Bits of 7 to 2 of register 0 should always be set to 0. Bit 1 is used in conjunction with 2 bits of register 1 to select the screen mode; it should be set to 1 for graphics mode 2, 0 for the other modes. Bit 0 is used to enable/disable input from an external VDP, and should normally be set to 0.

Register 1

Bit 7 of register 1 should always be set to 1. Bit 6 enables/disables the screen display, and should normally be set to 1. If you set it to 0, the screen will go blank. Bit 5 enables/disables VDP interrupts, and should normally be set to 1. Bits 4 and 3 are used to select the screen mode, in conjunction with bit 1 of register 0. Bit 4 has a value of 1 for text mode 0, 0 for the other modes. Bit 3 has a value of 1 for graphics mode 3, 0 for the other modes. Bit 2 should always be set to 0. Bit 1 is used to set the sprite size: 0 for 8×8 sprites, 1 for 16×16 sprites. Bit 0 selects the sprite magnification: 0 for unmagnified, 1 for magnified sprites.

BASIC SCREEN commands assign these values to registers 0 and 1:

screen mode	register 0	register 1
SCREEN 0	0	240
SCREEN 1,0	0	224
SCREEN 1,1	0	225
SCREEN 1,2	0	226
SCREEN 1,3	0	227
SCREEN 2,0	2	224
SCREEN 2,1	2	225
SCREEN 2,2	2	226
SCREEN 2,3	2	227
SCREEN 3,0	0	232
SCREEN 3,1	0	233
SCREEN 3,2	0	234
SCREEN 3,3	0	235

Register 2

Bits 7 to 4 of register 2 should be set to 0. Bits 3 to 0 represent the upper 4 bits of the 14 bit address of the start of the screen map. The possible values of the register, and corresponding addresses, are:

register 2	*start address of screen map in video RAM*	
	decimal	*hex*
0	0	0
1	1024	&H400
2	2048	&H800
3	3072	&HC00
4	4096	&H1000
5	5120	&H1400
6	6144	&H1800
7	7168	&H1C00
8	8192	&H2000
9	9216	&H2400
10	10240	&H2800
11	11264	&H2C00
12	12288	&H3000
13	13312	&H3400
14	14336	&H3800
15	15360	&H3C00

Register 3

The 8 bits of register 3 represent the upper 8 bits of the 14 bit colour table start address. The value of the register can be any number between 0 and 255. To determine the number to be assigned to the register, divide the colour table address by 64 (&H40).

Register 4

Bits 7 to 3 of this register should be set to 0. Bits 2 to 0 are the upper 3 bits of the 14 bit address of the pattern generator table (which contains the shapes of the text and graphics characters). The possible values and corresponding addresses are:

register 4	*start address of pattern generator table*	
	decimal	*hex*
0	0	0
1	2048	&H800
2	4096	&H1000
3	6144	&H1800
4	8192	&H2000
5	10240	&H2800
6	12288	&H3000
7	14336	&H3800

Register 5

Bit 7 of this register should be set to 0. The other 7 bits determine the start address of the sprite attribute table. The value of the register can be any number between 0 and 127. To determine the number to be assigned to the register, divide the sprite attribute table start address by 128 (&H80).

Register 6

Bits 7 to 3 of this register should be set to 0. The other 3 bits determine the start address of the sprite pattern table. For the possible values and corresponding start addresses, see the table given for register 4.

Register 7

Bits 7 to 4 of this register determine the foreground colour for the text modes, and bits 3 to 0 determine the background colour for all the screen modes. To calculate the number to be assigned to the register, use the formula:

register value = foreground colour code*16 + background colour code

Register 8

This is the read-only register. Bit 7 is an interrupt flag. Bit 6 is the fifth sprite flag, which is set to 1 when there are more than five sprites on a

horizontal line. Bit 5 is a sprite collision flag, which is set to 1 if two or more sprites overlap. Bits 4 to 0 give the number of the violating fifth sprite when bit 6 is set to 1. .

17.6 SAVING GRAPHICS ON TAPE

The information contained in Sections 17.1 to 17.5 is more useful than it may at first appear, to BASIC programmers as well as machine-code programmers. Here we will look at one application of it: saving a graphics screen on tape.

The ability to save graphics screens or sprite patterns on tape, without the programs which created them, would obviously be useful – but the BSAVE command can only be used to save sections of the main RAM; it does not work with the VRAM. However, you can copy sections of the VRAM into the main RAM, then save the copy. Which sections of the VRAM you decide to copy and save is up to you; in this example only the pattern generator and colour tables in mode 2 will be saved. This is sufficient to preserve a high-resolution picture which was created without the use of VPOKEs to the pattern name table.

The pattern generator table and colour table in mode 2 each occupy &H1800 bytes of memory. It is therefore necessary to reserve &H3000 bytes of the main memory for the copy, using the CLEAR command. Put at the start of your program:

```
10  CLEAR 255,&HC380
```

This will clear &HC380 to &HF37F in RAM.

The routine to save the picture is:

```
999    REM Picture save routine
1000  FOR  I=0  TO  &H17FF:POKE  &HC380+I,VPEEK(BA
      SE(11)+I):NEXT I
1010  FOR I=0 TO &H17FF:POKE &HDB80+I,VPEEK(BASE(12)
      +I):NEXT I
1020  SCREEN 0
1030  PRINT"Set cassette to RECORD, then press any key to
      continue":A$=INPUT$(1)
1040  BSAVE"CAS:PIC",&HC380,&HF37F
1050  END
```

Line 999: comment.

Line 1000: copies colour table into RAM.
Line 1010: copies pattern generator table into RAM.
Line 1020: selects text screen.
Line 1030: prints prompt, waits for keypress.
Line 1040: saves section of RAM on cassette.
Line 1050: end of routine.

The procedure can be reversed to reload the picture:

```
1999 REM Picture load routine
2000 CLEAR 255,&HC380
2010 BLOAD"CAS:PIC"
2020 SCREEN 2:CLS
2030 FOR I=0 TO &H17FF:VPOKE(BASE(11)+I),PEEK(&H
     C380+I):NEXT I
2040 FOR I=0 TO &H17FF:VPOKE(BASE(12)+I),PEEK(&HD
   B80+I):NEXT I
2050 GOTO 2050
```

Line 1999: comment.
Line 2000: clears space in RAM.
Line 2010: loads picture file into RAM.
Line 2020: selects graphics mode 2 and clears the screen.
Line 2030: copies colour table into VRAM.
Line 2040: copies pattern generator table into VRAM.
Line 2050: holds the display on the screen.

To test these routines, type in a routine to draw a picture in mode 2. You can use this one, or substitute your own (ensuring that the line numbers are between 100 and 990):

```
100 SCREEN 2
110 LINE(0,0)-(255,191),4,BF
120 LINE(8,8)-(247,183),10,BF
130 CIRCLE(128,96),80,8:PAINT(128,96),8,8
140 OPEN"GRP:" FOR OUTPUT AS #1
150 DRAW"BM60,90":PRINT #1,"Picture save test":CLOSE #1
```

When you RUN the completed program (line 10, the load and save routines and the picture drawing routine), the picture will be drawn, and then the prompt to set the cassette to RECORD will appear. Press any key and the picture will be recorded. Rewind the cassette and set it to PLAY, then enter RUN 2000 to load the picture again. You will see

a pattern of coloured blocks appear as the colour table is VPOKEd into the VRAM, then the details will be added.

This method of saving a screen is fairly slow. You can speed up the copying procedures greatly by the use of machine-code routines. There are BIOS routines you can call which will copy a section of the VRAM into RAM, and vice versa; all you have to do is to write two very short machine-code routines which load the appropriate values into the registers, then call these routines.

In both cases, register HL must hold the start address of the block to be moved, register DE must hold the start address of where it is to be moved to, and register BC must hold the length of the block. The routine to copy VRAM into RAM is called up by CALL &H0059 (CD 59 00), and the routine to copy RAM into VRAM is called up by CALL &H005C (CD 5C 00). On my computer BASE(11) returns the value &H2000, and BASE(12) returns the value &H0000. These are the machine-code routines (if BASE(11) and BASE(12) are different on your computer, substitute the appropriate numbers):

To copy VRAM into RAM:

```
LD HL,&H2000          21 00 20
LD DE,&HC380          11 80 C3
LD BC,&H1800          01 00 18
CALL &H0059           CD 59 00
LD HL,&H0000          21 00 00
LD DE,&HDB80          11 80 DB
LD BC,&H1800          01 00 18
CALL &H0059           CD 59 00
RET                   C9
```

To copy RAM into VRAM:

```
LD HL,&HC380          21 80 C3
LD DE,&H2000          11 00 20
LD BC,&H1800          01 00 18
CALL &H5C00           CD 5C 00
LD HL,&HDB80          21 80 DB
LD DE,&H0000          11 00 00
LD BC,&H1800          01 00 18
CALL &H5C00           CD 5C 00
RET                   C9
```

This initialization routine clears additional space for these routines,

loads them and defines their start addresses:

```
10 CLEAR 255,&HC300
20 FOR I=0 TO 24:READ A$:POKE &HC300+I,VAL("&H"+A
   $):NEXT I
30 DATA 21,00,20,11,80,C3,01,00,18,CD,59,00
40 DATA 21,00,00,11,80,DB,01,00,18,CD,59,00,C9
50 FOR I=0 TO 24:READ A$:POKE &HC340+I,VAL("&H"+A
   $):NEXT I
60 DATA 21,80,C3,11,00,20,01,00,18,CD,5C,00
70 DATA 21,80,DB,11,00,00,01,00,18,CD,5C,00,C9
80 DEF USR0=&HC300:DEF USR1=HC340
```

Replace lines 1000 and 1010 of the picture save routine with:

```
1000 X=USR0(0)
```

and replace lines 2000, 2030 and 2040 of the picture load routine with:

```
2030 X=USR1(0)
```

Remember to delete lines 1010, 2000 and 2040. You must make sure that the initialization routine has been run to load the machine code before using the new version of the load routine.

Exercise 17.1

Write a routine which will replace characters 0 to 3 with these characters:

```
-------*    *-------    ********    ********
------**    **------    *******-    -*******
-----***    ***-----    ******--    --******
----****    ****----    *****---    ---*****
---*****    *****---    ****----    ----****
--******    ******--    ***-----    -----***
-*******    *******-    **------    ------**
********    ********    *-------    -------*
```

Exercise 17.2

Write BASIC routines to save all 256 character patterns on tape, and reload them.

18

Summary of BASIC Keywords

18.1 INTRODUCTION

This chapter contains a brief summary of all the MSX BASIC keywords, with explanations of those which have not been introduced in earlier chapters.

In the syntax examples, round brackets ( ) are brackets which are required in the BASIC statement. Square brackets [ ] are used to denote optional parameters. Enclosed within broken brackets < > are descriptions of the type of parameter required, which should be replaced in use by the actual parameter.

The keywords are listed in alphabetical order, for ease of reference.

18.2 BASIC KEYWORDS

ABS returns the absolute value of a numeric constant, variable or expression.
Reference: Section 7.2.

AND a Boolean operator.
Reference: Section 7.1.

ASC returns the ASCII code number of the first character of its string argument.
Reference: Section 5.3.

ATN returns the arc-tangent of its argument in radians between –PI/2 and PI/2.
Reference: Section 7.2.

AUTO enables automatic line numbering.
Reference: Section 3.2.

BASE returns the current start address of a table in the video RAM.
Reference: Section 17.1.

BEEP produces a beep.
Reference: Section 12.2

BIN$ returns the binary equivalent of a decimal argument, in string form.
Reference: Section 6.1.

BLOAD loads a machine-code program or file saved using BSAVE. If followed by an R, the program auto-runs on loading.
Reference: Section 3.4.

BSAVE saves the contents of a specified section of memory, e.g. a machine-code program.
Reference: Section 3.3.

CALL calls an extended command contained in a RAM cartridge. Full instructions should be supplied with any cartridge which uses this command.
Syntax: CALL <extended command>[<list of arguments>]
Reference: –

CDBL converts an integer or single-precision number into a double-precision number.
Reference: Section 6.4.

CHR$ generates a character from its ASCII code. The argument must be an integer between 0 and 255.
Reference: Section 5.3.

CINT converts a single- or double-precision

number into an integer. Fractional parts are truncated. The argument must be between –32768 and 32767.

Reference: Section 6.4.

CIRCLE draws all or part of a circle or ellipse, in graphics modes 2 and 3 only.

Reference: Section 10.5.

CLEAR clears string storage space and memory space for machine-code routines.

Reference: Sections 6.2, 16.2.

CLOAD loads a BASIC program saved using CSAVE.

Reference: Section 3.4.

CLOAD? verifies a BASIC program, by comparing the version on cassette with the version in memory.

Reference: Section 3.3.

CLOSE closes an opened file channel.

Reference: Section 13.2.

CLS clears the screen to the current background colour.

Reference: Section 3.2.

COLOR sets the screen foreground, background and border colours.

Reference: Section 10.2.

CONT continues execution of a program halted by a STOP command or by <CTRL> and <STOP>.

Reference: Section 3.2.

COS returns the cosine of its argument. The argument must be in radians.

Reference: Section 7.2.

CSAVE saves a BASIC program in token format. You can specify baud rate 1 (1200 baud) or 2 (2400 baud).
Reference: Section 3.3.

CSNG converts an integer or double-precision number into a single-precision number.
Reference: Section 6.4.

CSRLIN returns the Y co-ordinate of the text cursor.
Syntax example: PRINT CSRLIN
Reference: –

DATA used to put data in a program, to be read by READ.
Reference: Section 8.2.

DEFDBL declares that any variable whose name begins with a letter within the specified range is double precision.
Reference: Section 6.2.

DEF FN used to define an extra function.
Reference: Section 7.6.

DEFINT declares that any variable whose name begins with a letter within the specified range is integer.
Reference: Section 6.2.

DEFSNG declares that any variable whose name begins with a letter within the specified range is single precision.
Reference: Section 6.2.

DEFSTR declares that any variable whose name begins with a letter within the specified range is a string variable.
Reference: Section 6.2.

DEF USR specifies the start address of a machine-code routine.
Reference: Section 16.3.

DELETE	deletes part of a program.
Reference:	Section 3.2.
DIM	dimensions an array.
Reference:	Section 6.3.
DRAW	draws a picture, as specified by a string of graphics macro language commands.
Reference:	Section 10.7.
ELSE	used as part of an IF<condition> THEN...ELSE structure, followed by the statements to be executed if the condition is not satisfied.
Reference:	Sections 8.1, 9.2
END	terminates execution of a program.
Reference:	Section 3.2.
EOF	end of file marker, returns –1 if end of file has been reached, 0 otherwise.
Reference:	Section 13.2.
EQV	a Boolean operator
Reference:	Section 7.1.
ERASE	erases an array.
Reference:	Section 6.3.
ERL	returns the number of the line in which an error has occurred.
Reference:	Section 15.1.
ERR	returns the error code when an error has been detected.
Reference:	Section 15.1.
ERROR	used to create an error.
Reference:	Section 15.1.
EXP	returns the natural exponential of its argument.
Reference:	Section 7.2.

FIX returns the integer part of its argument.

Reference: Section 7.2.

FOR sets up a loop.

Reference: Section 9.1.

FRE returns the amount of free memory space in the BASIC program area (with argument 0) or string area (with argument " ").

Syntax: FRE(<argument>)

Reference: –

GOSUB transfers execution to a specified subroutine.

Reference: Section 4.4.

GOTO transfers execution to a specified line number.

Reference: Section 4.3.

HEX$ gives the hexadecimal equivalent of a decimal integer, in string form.

Reference: Section 6.1.

IF used as part of an IF ... THEN ... ELSE structure, to test a condition.

Reference: Sections 8.1, 9.2.

IMP a Boolean operator.

Reference: Section 7.1.

INKEY$ returns the character of a depressed key (if any).

Reference: Section 8.1.

INP returns the byte read in at a specified input port.

Syntax: <num. var.>=INP(<port no.>)

Reference: –

INPUT inputs the value to be assigned to a variable.
Reference: Section 8.1.

INPUT# used to input data from a file.
Reference: Section 13.2.

INPUT$ inputs the specified number of characters from the keyboard or a file.
Syntax: <variable>=INPUT$(<no. of chars.>[,#<file no.>])
Reference: Section 8.1.

INSTR returns the position of a specified substring within a string.
Reference: Section 7.4.

INT rounds a number down to the nearest smaller integer.
Reference: Section 7.2.

INTERVAL ON enables time interval interrupts.
INTERVAL OFF disables time interval interrupts.
INTERVAL STOP holds time interval interrupts.
Reference: Section 11.3.

KEY used to assign a character string to a function key.
Reference: Section 11.1.

KEY LIST lists the contents of all the function keys.
Reference: Section 11.1.

KEY ON displays contents of function keys on bottom line of the screen.
KEY OFF erases this display.
Reference: Section 5.2.

KEY (n) ON enables interrupts by function key *n*.
KEY (n) OFF disables interrupts by function key *n*.
KEY (n) STOP holds interrupts by function key *n*.
Reference: Section 11.3.

LEFT$ returns the specified number of characters from the left of a string.
Reference: Section 7.4.

LEN returns the length of a string.
Reference: Section 7.4.

LET assigns a value to a variable (optional).
Reference: Section 6.2.

LINE draws a line, an outline box or a solid box.
Reference: Section 10.4.

LINE INPUT used to input a string which may include a comma.
Reference: Section 8.1.

LINE INPUT# used to input string data which may include commas etc. from a file.
Reference: Section 13.2.

LIST lists the BASIC program in memory on screen.
Reference: Section 3.2.

LLIST lists the program in memory on a printer.
Reference: Section 3.2.

LOAD loads a program saved using SAVE.
Reference: Section 3.4.

LOCATE moves the text cursor to the specified position.
Reference: Section 5.2.

LOG returns the natural logarithm of its argument.
Reference: Section 7.2.

LPOS returns the position of the printer head, as held in the printer buffer. This function requires a dummy argument.
Syntax: <num-var.>=LPOS(X).
Reference: —

LPRINT outputs the specified expression to a printer.
Reference: Section 5.2.

LPRINT USING outputs an expression to a printer, using the specified format.
Reference: Section 5.2.

MAXFILES sets the maximum number of files which can be used at any one time.
Reference: Section 13.1.

MERGE loads a program saved using SAVE, and merges it with a program already in memory.
Reference: Section 3.5.

MID$ returns the middle part of a given string, or replaces the middle part of a string with a specified string.
Reference: Section 7.4.

MOD gives the remainder after an integer division.
Reference: Section 7.1.

MOTOR toggles the cassette motor switch.
MOTOR ON switches the cassette motor on.
MOTOR OFF switches the cassette motor off.
Reference: Section 3.3.

NEW deletes the program in memory.
Reference: Section 3.2.

NEXT marks the end of a FOR ... NEXT loop.
Reference: Section 9.1.

NOT a Boolean operator.
Reference: Section 7.1.

OCT$ gives the octal equivalent of a decimal integer, in string form.
Reference: Section 6.1.

ON ERROR GOTO enables error trapping, and specifies the start of the error-handling routine.
Reference: Section 15.1.

ON ... GOSUB branches to one of a number of specified subroutines, depending on the value of an expression.
Reference: Section 9.2.

ON ... GOTO branches to one of a number of different lines, depending on the value of an expression.
Reference: Section 9.2.

ON INTERVAL GOSUB specifies the time interval and subroutine for time interval interrupts.
Reference: Section 11.3.

ON KEY GOSUB specifies the function key interrupt subroutines.
Reference: Section 11.3.

ON SPRITE GOSUB specifies the subroutine for sprite collision interrupts.
Reference: Sections 11.3, 14.6.

ON STOP GOSUB specifies the subroutine for <CTRL> and <STOP> interrupts.
Reference: Section 11.3.

ON STRIG GOSUB specifies the subroutines for space bar and joystick fire button interrupts.
Reference: Section 11.3.

OPEN opens a file, specifying the device, file name, mode and file channel number.
Reference: Section 13.2.

OR a Boolean operator.
Reference: Section 7.1.

OUT outputs 1 byte of data to a specified

output port.

Syntax: OUT<port no.>,<data>
Reference: –

PAD returns the status of a touch pad connected to one of the joystick ports. Arguments 0 to 3 return the status of a pad attached to port 1, and arguments 4 to 7 return the status of a pad attached to port 2.

Syntax: PAD(<integer between 0 and 7>)
Values returned: PAD(0)/PAD(4): 0 if pad is not touched
–1 if pad is touched
PAD(1)/PAD(5): X co-ordinate of touched location
PAD(2)/PAD(6): Y co-ordinate of touched location
PAD(3)/PAD(7): 0 if touch pad switch is pressed
–1 if touch pad switch is not pressed.

Reference: –

PAINT fills a closed shape surrounded by a border of a specified colour with a specified colour of paint.

Reference: Section 10.6.

PDL returns the value of a games paddle attached to one of the joystick ports. Up to twelve paddles can be used: paddles 1, 3, 5, 7, 9 and 11 in port 1, and paddles 2, 4, 6, 8, 10 and 12 in port 2. The value returned is between 0 and 255.

Syntax: PDL(<paddle no.>)
Reference: –

PEEK returns the contents of a specified memory location.

Reference: Section 16.2.

PLAY generates music, as specified by a string of

music macro language commands.
Reference: Section 12.3.

PLAY () tells you if music is being played by a specified channel.
Reference: Section 12.4.

POINT returns the colour code of a specified screen point.
Reference: Section 10.3.

POKE writes data to a specified memory location.
Reference: Section 16.2.

POS returns the X co-ordinate of the text cursor. It needs a dummy argument.
Syntax: POS(<any integer>)
Reference: –

PRESET resets a point on the graphics screen to the background colour, or sets it to a specified colour.
Reference: Section 10.3.

PRINT prints an expression on the text screen.
Reference: Section 5.2.

PRINT USING prints data on the text screen, using the specified format.
Reference: Section 5.2.

PRINT# writes data to a specified file.
Reference: Section 13.2.

PRINT # USING writes data to a file, using the specified format.
Reference: Section 13.2.

PSET sets a point on the graphics screen to the foreground colour, or a specified colour.
Reference: Section 10.3.

PUT SPRITE displays a sprite on the screen. The sprite plane, pattern number, co-ordinates and colour can be specified.
Reference: Section 14.3.

READ reads data specified in a DATA statement and assigns it to a variable.
Reference: Section 8.2.

REM used to insert a comment in a program.
Reference: Section 4.5.

RENUM renumbers the lines of a program.
Reference: Section 3.5.

RESTORE specifies the DATA statement to be used by the next READ statement. Enables data to be read more than once.
Reference: Section 8.2.

RESUME returns execution to main program after execution of an error-trapping routine.
Reference: Section 15.1.

RETURN returns execution from the end of a subroutine to the statement following the GOSUB statement.
Reference: Section 4.4.

RIGHTS$ returns the specified number of characters from the right of a string.
Reference: Section 7.4.

RND generates a random number between 0 and 1.
Reference: Section 7.3.

RUN runs a program. The start line may be specified.
Reference: Section 3.2.

SAVE saves a BASIC program in ASCII format.
Reference: Section 3.3.

SCREEN selects the screen mode, sprite size, key click/no key click, cassette interface baud rate (1 for 1200 baud, 2 for 2400 baud) and printer type (0 for MSX printer, between 1 and 255 for printer which cannot handle MSX graphics characters).

Syntax: SCREEN[<mode>],[<sprite size>],[<key click switch>],[<baud rate>],[<printer type>]

Reference: Sections 5.1,10.1,12.1,14.1

SGN returns the sign of its argument: –1 if argument is negative, 0 if argument is 0, 1 if argument is positive.

Reference: Section 7.2.

SIN returns the sine of its argument. The argument must be in radians.

Reference: Section 7.2.

SOUND writes data to the programmable sound generator registers, to generate sound effects.

Reference: Section 12.6.

SPACE generates a string containing the specified number of spaces.

Syntax: SPACE$(<no. of spaces>)

Reference: –

SPC used in PRINT statements, to print spaces.

Reference: Section 5.2.

SPRITE ON enables sprite collision interrupts.
SPRITE OFF disables sprite collision interrupts.
SPRITE STOP holds sprite collision interrupts.
Reference: Section 11.3, 14.6

SPRITE$ defines a sprite pattern.
Reference: Section 14.2.

SQR	returns the square root of its (positive) argument.
Reference:	Section 7.2.
STEP	used to specify the counter increment in a FOR … NEXT loop.
Reference:	Section 9.1.
STICK	returns the direction indicated by the cursor keys or a joystick.
Reference:	Section 11.2.
STOP	stops execution of a program (which can then be restarted by CONT).
Reference:	Section 3.2.
STOP ON	enables <CTRL> and <STOP> interrupts.
STOP OFF	disables <CTRL> and <STOP> interrupts.
STOP STOP	holds <CTRL> and <STOP> interrupts.
Reference:	Section 11.3.
STR$	converts its numeric argument into a string.
Reference:	Section 6.4.
STRIG	returns the status of the space bar or a joystick fire button (–1 if pressed, 0 if not pressed).
Reference:	Section 11.2.
STRIG ON	enables space bar and fire button interrupts.
STRIG OFF	disables space bar and fire button interrupts.
STRIG STOP	holds space bar and fire button interrupts.
Reference:	Section 11.3.
STRING$	returns a string with a specified number of specified characters.
Reference:	Section 5.3.

SWAP swaps the contents of two variables.
Reference: Section 7.5.

TAB used in a PRINT statement to move the cursor to the specified column.
Reference: Section 5.2.

TAN returns the tangent of its argument. The argument must be in radians.
Reference: Section 7.2.

THEN used as part of an IF<condition>THEN...ELSE structure, followed by the statements to be executed if the condition is satisfied.
Reference: Sections 8.1, 9.2.

TIME returns the value of the built-in timer.
Reference: Section 7.3.

TO used to specify the end value of the counter in a FOR...NEXT loop.
Reference: Section 9.1.

TROFF switches off the trace facility.
Reference: Section 15.2.

TRON switches on the trace facility.
Reference: Section 15.2.

USR calls a machine-code subroutine defined using DEFUSR. The argument is a parameter to be passed to the machine-code routine (or a dummy); the function returns a parameter from the machine-code routine.
Reference: Section 16.3.

VAL returns the value of a string which starts with a number.
Reference: Section 6.4.

VARPTR returns the start address in memory where the data assigned to a specified variable is stored.
Syntax: VARPTR(<variable>)
Reference: –

VDP used to access the video display processor registers: to write to registers 0 to 7, and to read register 8.
Reference: Section 17.5.

VPEEK gives the contents of a specified address in the video RAM.
Reference: Section 17.1.

VPOKE writes data to a specified address in the VRAM.
Reference: Section 17.1.

WAIT suspends program execution until the input from a specified port reaches a specified value, i.e. until (<input> XOR <expression 2>) AND <expression 1> <>0
Syntax: WAIT<port no.>,<expression 1>[,<expression 2>]
Reference: –

WIDTH sets the width of the text screen.
Reference: Section 5.1.

XOR a Boolean operator.
Reference: Section 7.1.

19
Some BASIC Programs

19.1 INTRODUCTION

This chapter describes the planning and writing of three BASIC programs: a joystick drawing program, a sprite designer and a simple music synthesizer. The listings will be useful additions to your program library – but it is the writing process which is most important, not the programs themselves. Once you have mastered all the techniques used here, you should be able to tackle your own programming projects with confidence.

The process has three main stages: first, deciding precisely what the program is to do, second, breaking the task down into small stages and drawing up a flowchart, and third, writing the routines which make up the program. As you gain more experience in programming you may find that, for programs of this length, you can do much of the preliminary work in your head instead of putting it all down on paper. However, it is still useful to know how the job should be tackled, and full documentation of the planning process can be a great help if the program later requires revision.

19.2 JOYSTICK DRAWING PROGRAM

This program is intended to enable you to draw pictures with a joystick. The idea is that when you point the joystick in a particular direction, a line will be drawn in that direction.

The high-resolution graphics mode (mode 2) will be used, as this is the best mode for drawing detailed pictures. It would be nice to be able to select the background colour; in mode 2, the screen background colour can only be changed by clearing the screen, so a routine will be

included to increase the background colour code by 1 (resetting it to 1 when it exceeds 15), and clear the screen, whenever function key 1 is pressed. Changing the foreground colour does not necessitate clearing the screen; another routine will increase the foreground colour code by 1 (and again reset it to 1 when it exceeds 15) whenever function key 2 is pressed. Function key interrupts can be used for both these routines.

A means of changing the current drawing position without actually drawing a line would be useful. This could be provided by drawing in the background colour whenever the joystick fire button is depressed. But it will then be necessary to have some means of identifying the current draw position: a cross-shaped cursor, in the current foreground colour. A sprite is an obvious choice for this.

An instruction routine will obviously be necessary, to explain to the user how to operate the program. The instructions will remain on the screen until a key is pressed.

It is possible to add lots of other facilities to a program of this type. You could include routines to draw circles, or to paint areas of the screen, or incorporate the routine given in Section 17.6 to save pictures on tape. However, I will leave you to add these extras to the basic program described here.

The program stages are:

1. Display instructions on screen, wait for keypress.
2. Specify function key interrupt routines.
3. Specify initial values of background colour code (B), foreground colour code (F), cursor X co-ordinate (X), cursor Y co-ordinate (Y).
4. Enable interrupts from function keys 1 and 2.
5. Select graphics mode 2 with 8×8 unmagnified sprites, and clear the screen.
6. Specify sprite pattern for cursor.
7. Display cursor at (X,Y) in colour F.
8. Set draw colour to B if fire button is depressed, F otherwise.
9. Determine direction in which joystick is pointing. If joystick is centred, go back to stage 8.
10. Branch to routine to draw line from centre of cursor in direction in which joystick is pointing, checking to see that the new draw position is on the screen, and increment or decrement X and/or Y as appropriate.
11. Go back to stage 7.
12. Function key 1 interrupt routine: increment B, if B>15 then reset B to 1. Clear the screen. Return to stage 7.
13. Function key 2 interrupt routine: increment F, if F>15 then reset

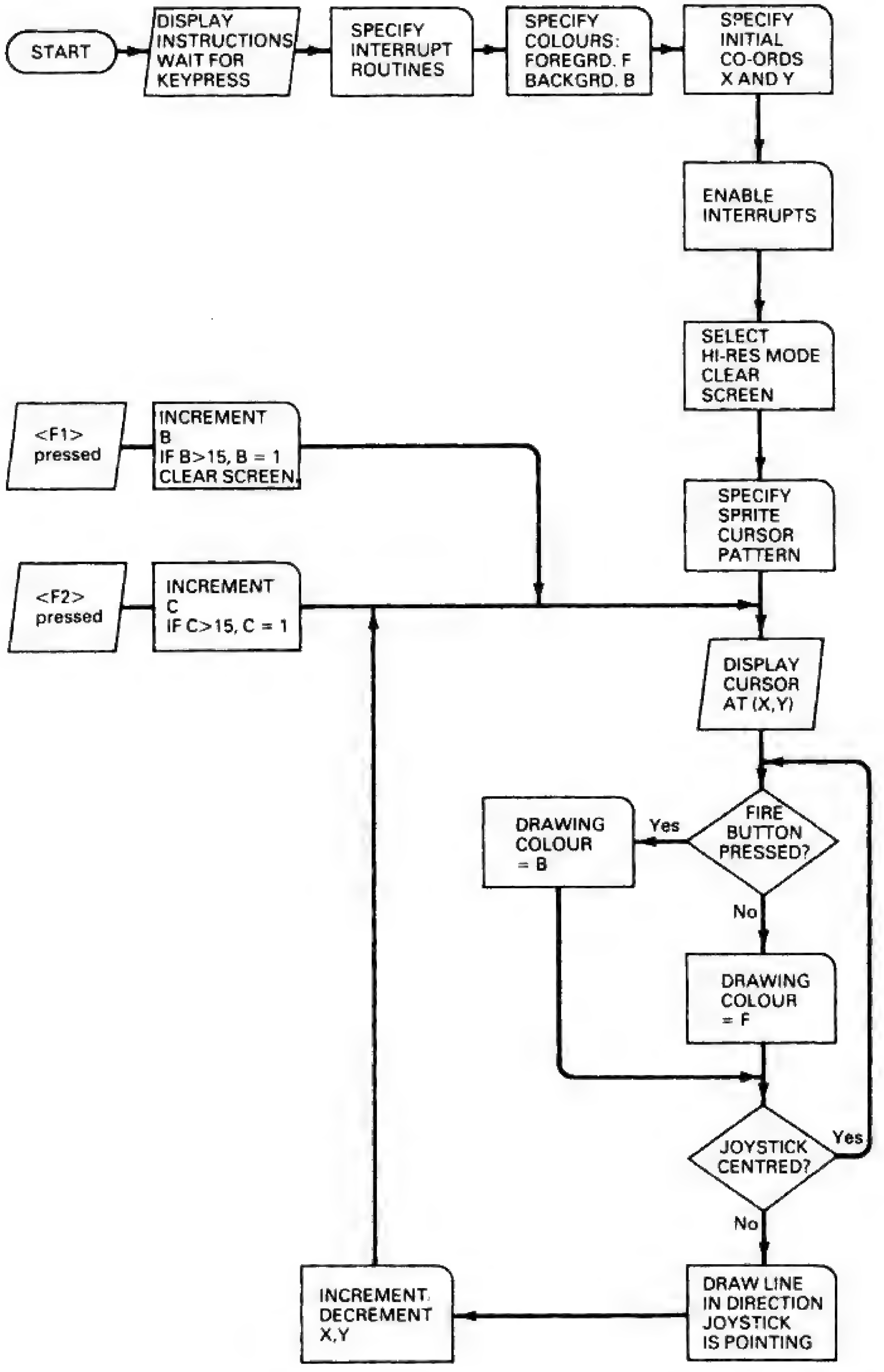


Figure 12 *Joystick drawing flowchart*

F to 1. Return to stage 7.

The flowchart for this program is shown in Figure 12.

The variables to be used are:

B	screen background colour
F	screen foreground (ink) colour
X	X co-ordinate of cursor
Y	Y co-ordinate of cursor

The program starts with comments giving its title etc.:

```
10 REM Joystick Drawing Program
20 REM By Margaret Norman
```

Now the routine for stage 1:

```
30 REM Instructions
40 CLS:PRINT“   JOYSTICK DRAWING”:PRINT
50 PRINT“Move the joystick to draw a picture.”
60 PRINT“Hold the fire button down to draw”
70 PRINT“in the background colour.”
80 PRINT“Press F1 to change the background”
90 PRINT“colour and clear the screen.”
100 PRINT“Press F2 to change the ink colour.”
110 PRINT“Press any key to start”:A$=INPUT$(1)
```

Stage 2:

```
120 REM Specify interrupt routines:
130 ON KEY GOSUB 450,490
```

Stage 3:

```
140 REM Initialization of variables
150 B=15:F=1:X=100:Y=100
```

Stage 4, enabling the interrupts (note that the interrupt routines have already been specified):

```
160 REM Enable interrupts
170 KEY(1) ON:KEY(2) ON
```

Stage 5:

```
180  REM Select mode 2, clear screen
190  SCREEN 2,0: COLOR F,B:CLS
```

Now that the sprite size has been specified, the sprite pattern can be defined (stage 6):

```
200  REM Define sprite pattern
210  SPRITE$(0)=CHR$(129)+CHR$(66)+CHR$(36)
     +CHR$(24)+CHR$(24)+CHR$(36)+CHR$(66)+CHR$(129)
```

Stage 7, the start of the main program loop:

```
220  REM Display cursor
230  PUT SPRITE 0, (X,Y),F,0
```

Stage 8. (Note that value of joystick 1 fire button 1 is tested. If you do not have a joystick, change STRIG(1) in line 250 to STRIG(0) to use the space bar/cursor keys.)

```
240  REM Set drawing colour
250  IF STRIG(1)=0 THEN DRAW"C=F;" ELSE DRAW"C=B;"
```

Stages 9, 10 and 11 are combined into one routine. Note that the DRAW commands move the draw position from the top left-hand corner of the cursor to the middle of the cursor and back again. (To use cursor keys, change STICK(1) in line 270 to STICK(0).)

```
260  REM Read joystick position, draw line
270  ON STICK(1)=1 GOTO 250,280,300,320,340,360,380,400,420
280  IF Y>-4 THEN DRAW"BF4U1BH4":Y=Y-1
290  GOTO 230
300  IF X<251 AND Y>-4 THEN DRAW"BF4E1BH4"
     :X=X+1:Y=Y-1
310  GOTO 230
320  IF X<251 THEN DRAW"BF4R1BH4"
     :X=X+1
330  GOTO 230
340  IF X<251 AND Y<187 THEN DRAW"BF4F1BH4"
     :X=X+1:Y=Y+1
350  GOTO 230
```

```
360 IF Y<187 THEN DRAW"BF4D1BH4":Y=Y+1
370 GOTO 230
380 IF X>-4 AND Y<187 THEN DRAW"BF4G1BH4"
    :X=X-1:Y=Y+1
390 GOTO 230
400 IF X>-4 THEN DRAW"BF4L1BH4":X=X-1
410 GOTO 230
420 IF X>-4 AND Y>-4 THEN DRAW"BF4H1BH4"
    :X=X-1: Y=Y-1
430 GOTO 230
```

The interrupt routines (stages 12 and 13) are very short:

```
440 REM F1 interrupt routine
450 B=B+1:IF B>15 THEN B=1
460 COLOR F,B:CLS
470 RETURN 230
480 REM F2 interrupt routine
490 F=F+1:IF F>15 THEN F=1
500 RETURN 230
```

Figure 13 shows a picture drawn with this program. (Not a very good one – I'm a writer, not an artist!)

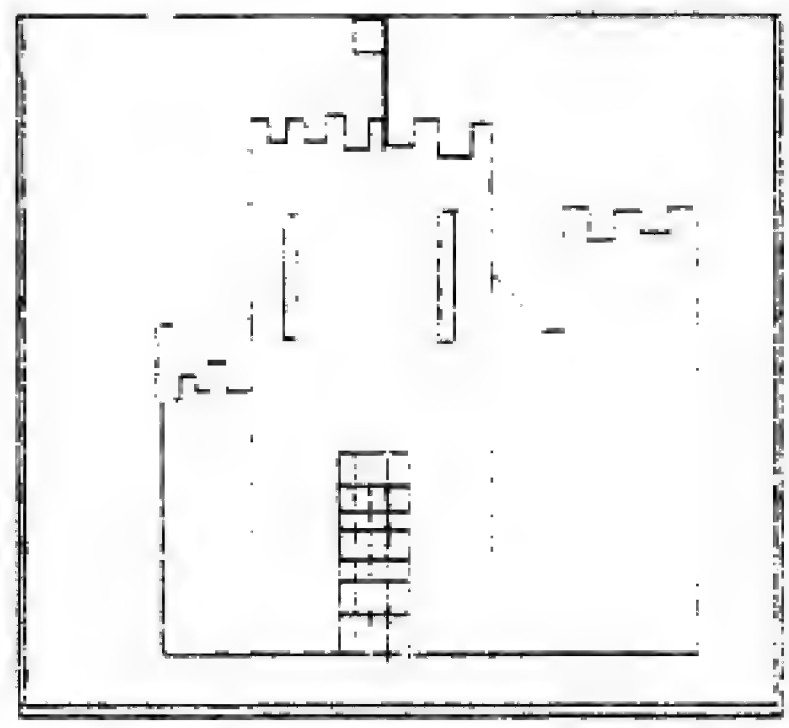


Figure 13 *Picture drawn with joystick drawing program*

19.3 SPRITE DESIGNER

This program is intended to simplify the process of designing sprites (and user-defined graphics characters). It must provide a grid of squares on the screen, a simple means of filling in selected squares to form the

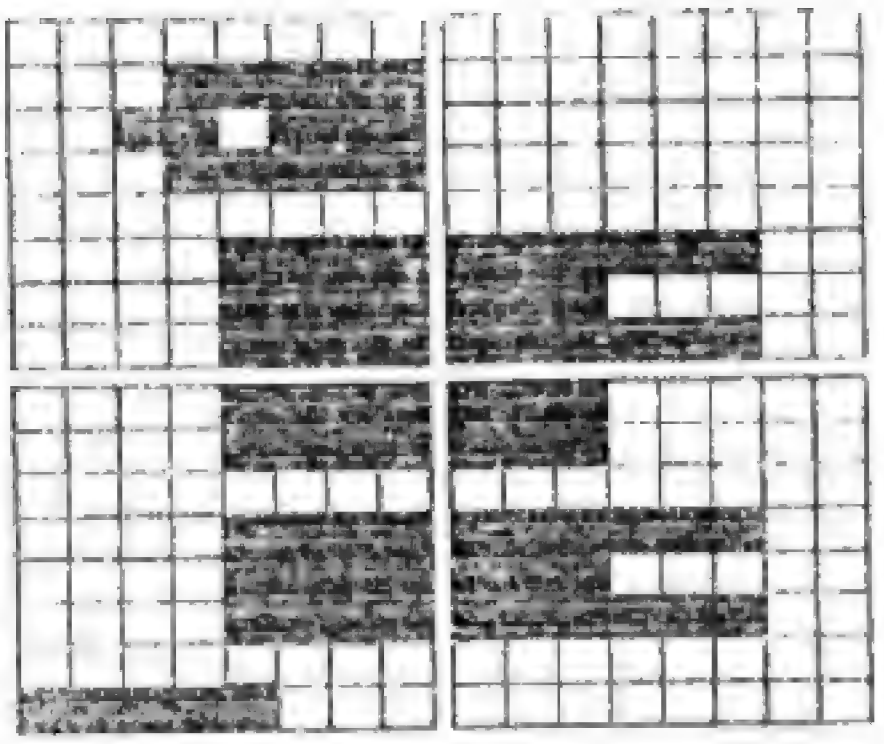


Figure 14 *Designing a sprite*

sprite design, and a means of printing out the sprite data.

It would be convenient if the same program could be used for designing 8×8 and 16×16 sprites. This can easily be arranged, by using a grid of 16×16 squares, clearly divided into four 8×8 square blocks (see Figure 14.) A single block can be used to design an 8×8 sprite, or all four blocks can be used to design a 16×16 sprite – or the blocks can be used individually, to design four 8×8 sprites at once.

How can the squares to be filled in be selected? A cursor which can be moved from one square to another is an obvious choice. A small square sprite will be used, coloured so that it can be distinguished easily when it is positioned over a filled square. Function key interrupt routines can be used for filling in squares, and for blanking squares which have been filled in error.

It would be useful if the designed sprite could be displayed on the screen actual size, as a pattern of squares is not always a very good guide to what the finished sprite will look like. The sprite data will obviously have to be calculated before the sprite can be displayed. If a subroutine is used to calculate the sprite data, the same routine can be called when a request is made for the data to be printed out. As the subroutine will take some time to execute, a flag can be used to indicate whether the data for the current design has already been calculated. This flag can be set to 1 when the subroutine is called, and reset to 0 when the routines to fill and blank squares are called. Two more function key interrupt routines can be used to display the sprite and to print out the data.

Again, lots more facilities could be included in the program, but adding further refinements will be left as an exercise for you.

The program stages are:

1. Display instructions, wait for keypress.
2. Specify function key interrupt routines (four routines: <F1> to fill

squares, <F2> to blank squares, <F3> to display the sprite and <F4> to print out the data).

3. Dimension array S(3,7) to contain the sprite data.
4. Select high-resolution screen with 16×16 unmagnified sprites, background white, foreground black, and clear the screen.
5. Define the pattern for the sprite cursor.
6. Draw the grid.
7. Enable the function key interrupts.
8. Specify initial values of the cursor X co-ordinate (X) and Y co-ordinate (Y).
9. Display the cursor at co-ordinates (X,Y).
10. Determine direction indicated by the cursor keys and branch to routine to calculate new values of X and Y, ensuring that the cursor position remains with the grid and allowing for the gaps between the blocks of squares.
11. Return to stage 9.
12. <F1> interrupt routine: paint the square containing the point (X,Y) black, set flag to 0, return to stage 9.
13. <F2> interrupt routine: put a solid white square within the square containing the point (X,Y), set flag to 0, return to stage 9.
14. <F3> interrupt routine: call subroutine to calculate sprite data, define sprite pattern, display sprite, return to stage 9.
15. <F4> interrupt routine: call subroutine to calculate sprite data, select text screen, print out data. Print prompt; if key E is pressed, end of program; if key S is pressed, set flag to 0 and return to stage 4.
16. Subroutine to calculate sprite data: if flag=1 then return, else calculate sprite data, set flag to 1, return.

Figure 15 shows the flowchart.

The variables to be used are:

I,J,K	loop counters
D	delay loop counter
X	X co-ordinate of cursor
Y	Y co-ordinate of cursor
S(I,J)	sprite data
F	flag
A$	input variable

The usual introductory comments are followed by the instructions (stage 1):

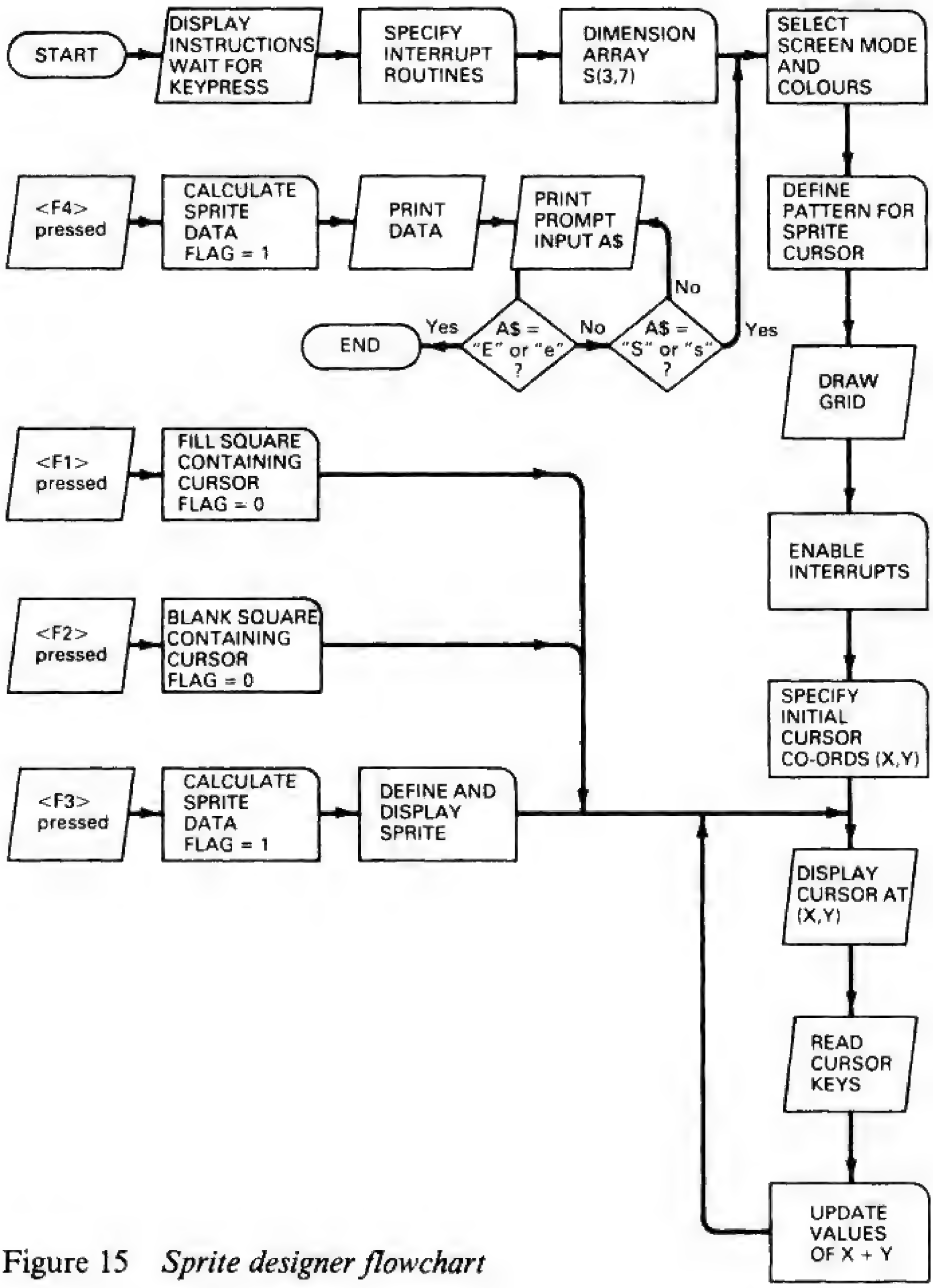


Figure 15 *Sprite designer flowchart*

```
10  REM Sprite designer
20  REM by Margaret Norman
30  REM Instructions
40  CLS:PRINT"    SPRITE DESIGNER":PRINT
50  PRINT"Move the orange cursor with the    cursor keys."
60  PRINT"Press F1 to fill in a square."
70  PRINT"Press F2 to blank a square."
```

```
 80 PRINT"Press F3 to display the sprite."
 90 PRINT"Press F4 to print the sprite data."
100 PRINT"Press any key to start":A$=INPUT$(1)
```

Stages 2, 3 and 4 are very simple to program:

```
110 REM Specify interrupt routines
120 ON KEY GOSUB 500,600,700,800
130 REM Dimension array to hold sprite data
140 DIM S(3,7)
150 REM Set up and clear screen
160 SCREEN 2,2:COLOR 1,15,1:CLS
```

In stage 5, the cursor sprite data is VPOKEd into the VRAM:

```
170 REM Sprite cursor data
180 FOR I=0 TO 31:VPOKE BASE(14)+I,0:NEXT I
190 FOR I=2 TO 5: VPOKE BASE(14)+I,&H3C:NEXT I
```

Now to draw the grid (stage 6):

```
200 REM Draw grid
210 FOR K=0 TO 84 STEP 84
220 FOR I=20 TO 100 STEP 10
230 LINE (I+K,10)-(I+K,90),1:LINE(I+K,94)-(I+K,174),1:
    NEXT I
240 FOR J=10 TO 90 STEP 10
250 LINE(20,J+K)-(100,J+K),1:LINE(104,J+K)
    -(184,J+K),1:NEXT J
260 NEXT K
```

Stages 7 and 8:

```
290 REM Enable interrupts
300 FOR I=1 TO 4:KEY(I) ON:NEXT I
310 X=22:Y=12
```

Stage 9 is the start of the main program loop:

```
320 REM Display cursor
330 PUT SPRITE 0,(X,Y),8,0
```

The routine for stages 10 and 11 does not permit diagonal movement of the cursor. A delay loop is put in to slow the movement down:

```
340 REM Movement routine
350 FOR D=0 TO 100:NEXT D
360 ON STICK(0)+1 GOTO 330,370,330,390,330,410,330,
    430,330
370 IF Y>12 THEN Y=Y-10:IF Y=86 THEN Y=82
380 GOTO 330
390 IF X<176 THEN X=X+10:IF X=102 THEN X=106
400 GOTO 330
410 IF Y<166 THEN Y=Y+10: IF Y=92 THEN Y=96
420 GOTO 330
430 IF X>22 THEN X=X-10:IF X=96 THEN X=92
440 GOTO 330
```

Stages 12 and 13, the first two interrupt routines:

```
490 REM F1 interrupt routine
500 PAINT(X,Y),1,1:F=0
510 RETURN 330
590 REM F2 interrupt routine
600 LINE(X-1,Y-1)-(X+7,Y+7),15,BF:F=0
610 RETURN 330
```

Stage 14 calls the sprite data calculation subroutine. As this takes some time to execute, there is a beep to let you know when it is finished:

```
690 REM F3 interrupt routine
700 GOSUB 900
710 FOR I=0 TO 3:FOR J=0 TO 7
720 VPOKE BASE(14)+32+I*8+J,S(I,J):NEXT J,I
730 PUT SPRITE 1, (230,100),1,1:BEEP
740 RETURN 330
```

Stage 15, the final interrupt routine:

```
790 REM F4 interrupt routine
800 GOSUB 900
810 SCREEN 0:CLS:PRINT"SPRITE DATA"
820 FOR I=0 TO 3: FOR J=0 TO 7
830 PRINT S(I,J);:NEXT J:PRINT:NEXT I
```

```
840 PRINT"Press 'S' to restart or 'E' to end program."
850 A$=INPUT$(1)
860 IF A$="E" OR A$="e" THEN END
870 IF A$="S" OR A$="s" THEN F=0:RETURN 160
880 GOTO 850
```

Finally, the subroutine to calculate the sprite data (stage 16):

```
890 REM Subroutine to calculate sprite data
900 IF F=1 THEN RETURN
910 FOR I=0 TO 3
920 FOR J=0 TO 7
930 V=10*(J+1)-84*(I=1 OR I=3):S(I,J)=0
940 FOR K=0 TO 7
950 H=10*(K+2)-84*(I=2 OR I=3)
960 S(I,J)=S(I,J)-((POINT(H+1,V+2)=1)*2^(7-K))
970 NEXT K,J,I
980 F=1
990 RETURN
```

19.4 MUSIC SYNTHESIZER

Now to turn the computer into a musical instrument. This program will enable you to use the computer keyboard like a piano keyboard.

As there are four rows of keys on the keyboard, we will provide a two-octave range, the bottom two rows of keys being used for the white and black notes in the first octave, and the top two rows of keys being used for the second octave. To indicate which key represents which note, the screen display will show a piano keyboard with the appropriate characters marked on the keys.

The PLAY command plays notes of a predetermined length; if this command is used in the program, it will only be possible to play notes of one length. With the SOUND command, however, you can produce a note which will continue to sound until it is switched off; this command is much more appropriate for a program of this type. If the note for each key is to continue until another key is pressed, it will be necessary to press a key which has no note associated with it to stop the sound.

The high and low bytes of the frequency for each note can be calculated in advance, then put in DATA statements within the

program and read into an array.

The program stages are:

1. Dimension arrays to hold the frequency data, and read the data into these arrays.
2. Draw the keyboard.
3. Set the sound generator to emit a tone from channel A.
4. Define a string which contains all the characters which correspond to notes.
5. Input a character, A$, from the keyboard.
6. If the character A$ does not correspond to a note, set channel A volume to 0 and return to stage 5.
7. Set channel A volume to 8 and set channel A frequency to produce the note corresponding to A$, then return to stage 5.

The flowchart is shown in Figure 16.

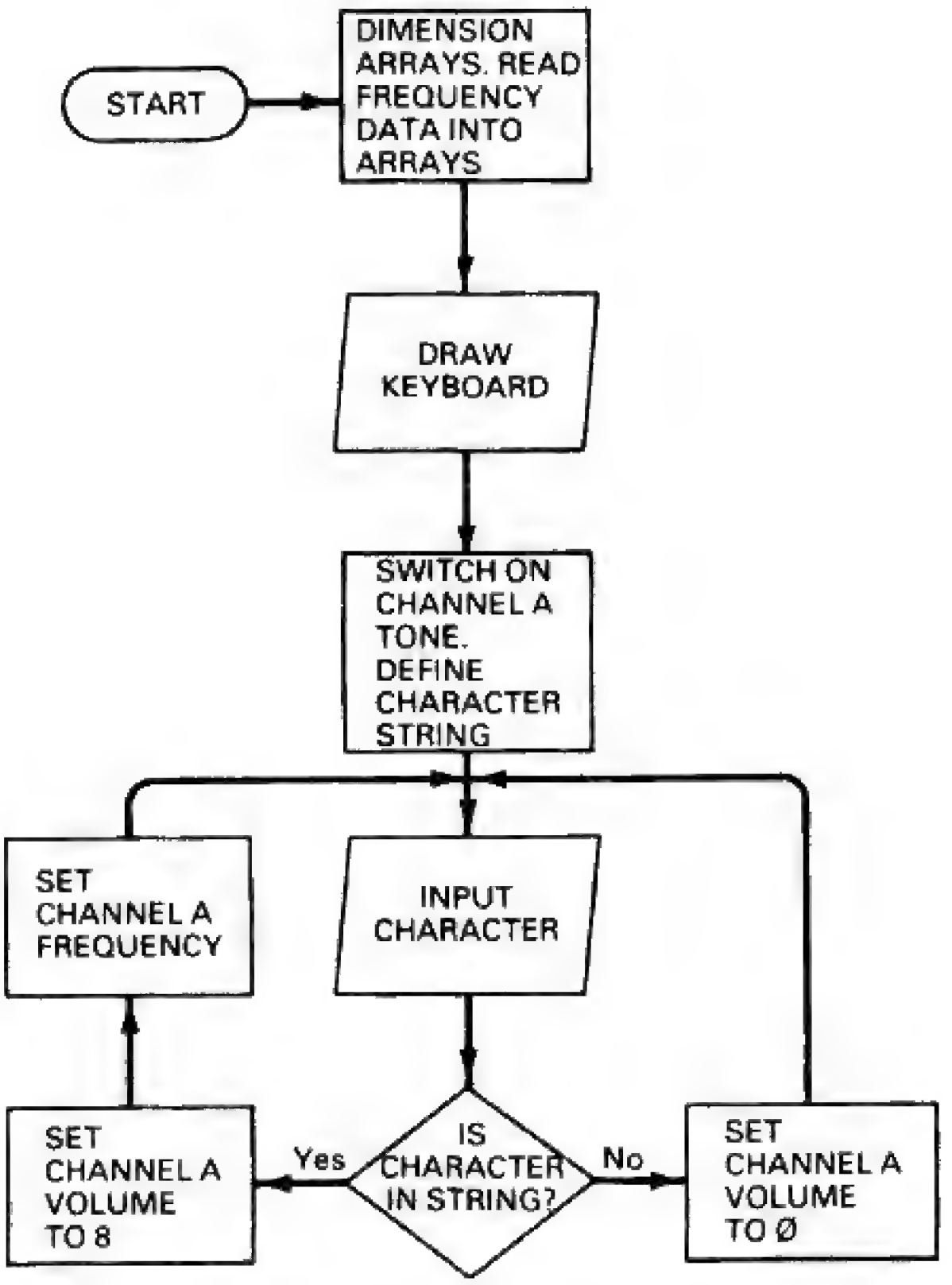


Figure 16 *Music synthesizer flowchart*

The variables to be used are:

I,J	loop counters
H(I)	high frequency for note I
L(I)	low frequency for note I
K$	string containing the characters corresponding to notes
A$	input variables
N	number of note to be played

The introductory comments and stage 1 (the data is for a computer with clock frequency 1996750 Hz – see Chapter 12):

```
10  REM Music synthesizer
20  REM By Margaret Norman
30  DIM H(25),L(25)
40  REM Read in frequency data
50  FOR I=1 TO 25: READ H(I),L(I):NEXT I
60  DATA 3,208,3,141,3,101,3,53,3,5,2,219,2,178,2,139,
      2,102
70  DATA 2,68,2,36,2,5,1,232,1,204,1,179,1,154,1,131,
      1,110
80  DATA 1,89,1,70,1,51,1,34,1,18,1,2,0,244
```

Stage 2, which draws the keyboard shown in Figure 17:

```
100  REM Draw keyboard
110  SCREEN 2,,0:REM key click off
120  COLOR 8,15,15:CLS
130  FOR I=1 TO 15
140  LINE(16*I,50)-(16*(I+1),150),1,B:NEXT I
150  FOR J=1 TO 13
160  IF J=3 OR J=7 OR J=10 THEN 180
170  LINE(16*J+12,50)-(16*J+24,120),1,BF
180  NEXT J
190  OPEN "GRP:" FOR OUTPUT AS #1
200  PRESET(22,130):PRINT #1,"Z X C V B N M   W E R T Y U
     I"
210  PRESET(134,124):PRINT   #1,",":PRESET(134,136):PRINT
     #1,"Q"
220  PRESET(32,90):PRINT #1,"S D   G H J   2 3   5 6 7"
230  CLOSE #1
```

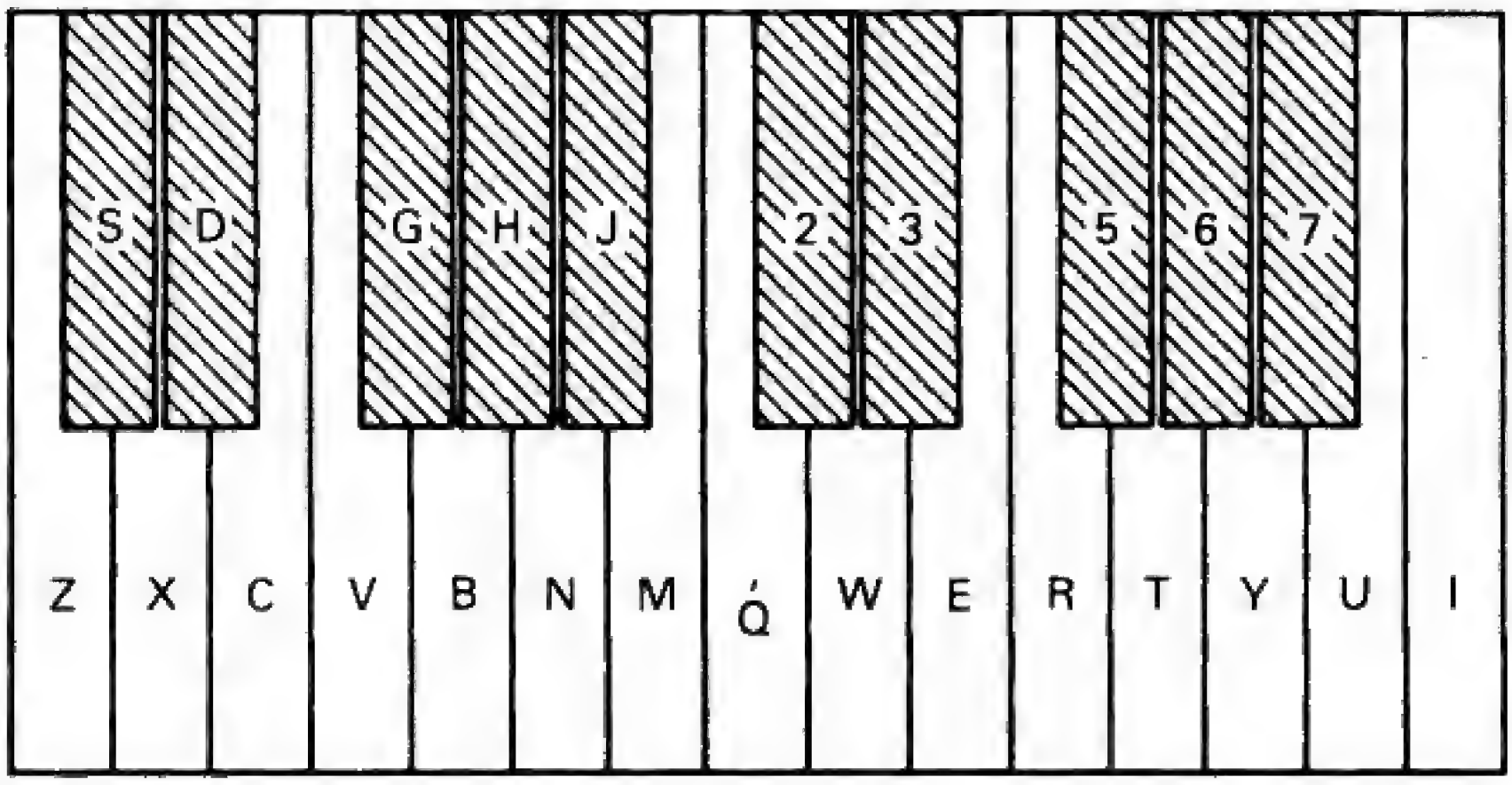


Figure 17 *The keyboard screen display*

Stage 3:

```
240  SOUND 7,254:REM Channel A tone
```

Stage 4. Note that this string does not contain the SHIFTed characters:

```
250  REM Define string
260  K$="zsxdcvgbhnjm,q2w3er5t6y7ui"
```

Stage 5, the start of the main program loop:

```
270  REM Main routine
280  A$=INPUT$(1)
```

Stages 6 and 7

```
290  N=INSTR(K$,A$)
300  IF N=0 THEN SOUND 8,0:GOTO 280
310  IF N>13 THEN N=N-1
320  SOUND 8,8:SOUND 0,L(N):SOUND 1,H(N)
330  GOTO 280
```

This is a short but effective progam. Again, you could easily add further refinements – facilities to vary the volume or increase the range, or to select different volume envelopes, for example.

Answers to Exercises

4.1 10,20,80,90,30,60,70,40,50,100,130.

5.1
```
10 REM Letter heading
20 LPRINT"Tel 01–234–5678"SPC(16)"1 New Road"
30 LPRINT TAB(33)"Any Town"
40 LPRINT TAB(35)"Blankshire"
50 LPRINT TAB(37)"England"
60 END
```

5.2
```
10 REM Table
20 CLS
30 PRINT TAB(4)"n"TAB(13)"n^2"TAB(23)
   "n^3"TAB(33)"n^4"
40 N=1:GOSUB 100
50 N=2:GOSUB 100
60 N=3:GOSUB 100
70 N=4:GOSUB 100
80 N=5:GOSUB 100
90 END
100 PRINT TAB(3)N;
110 PRINT USING "##########";N^2,N^3,N^4
120 RETURN
```

6.1 (a) 572=&B1000111100=&O1074=&H23C
(b) 3144=&B110001001000=&O6110=&HC48
(c) -2197=&B1111011101101011=&O173553=&HF76B

6.2 (a) &B101100010=354
(b) &O374=252
(c) &HD2C=3372

6.3 &H4E=&B1001110

6.4 (a) and (d).

6.5 (a) Integer
(b) String
(c) Single precision

6.6 (a) The CLEAR command in line 20 undimensions the array, so a Subscript out of range error will occur in line 30.
(b) The array B(I) is dimensioned twice.
(c) VAL returns a number, which cannot be assigned to the string variable B$.
(d) String variables cannot exceed 255 characters.

6.7 (a) –85
(b) "294"
(c) –876

7.1 –13

7.2
```
10 N$="Margaret Norman"
20 S=INSTR(N$," ")
30 F$=LEFT$(N$,S-1)
40 PRINT "Hello ";F$
50 END
```
(Substitute your own name in line 10.)

7.3 88725

8.1
```
10 CLS:N$=""
20 LOCATE 17,11
30 A$=INPUT$(1)
40 PRINT A$=;:N$=N$+A$
50 IF LEN(N$)<5 THEN GOTO 30
60 END
```

8.2
```
10 CLS
20 PRINT"Question 5. Which of these commands automatically prints a question mark?"
30 PRINT"(1) INPUT"
40 PRINT"(2) LINE INPUT"
50 PRINT"(3) INPUT$"
60 PRINT"(4) INKEY$"
70 INPUT"Answer 1, 2, 3 or 4";A
80 IF A=1 THEN PRINT "Well done!" ELSE PRINT"You had better re-read this chapter"
90 END
```

8.3
```
10 CLS
20 READ X,Y,C
30 IF X=-1 THEN END
40 LOCATE X,Y:PRINT CHR$(C);
50 GOTO 20
60 DATA 10,10,205,6,14,207,14,14,208,10,18,206,-1,0,0
```

9.1
```
10 SCREEN 1:CLS
20 FOR N=0 TO 31:PRINT CHR$(1)+CHR$(N+64)
   ;:NEXT N
```

```
    30 FOR N=32 TO 255:PRINT CJHR$(N);:NEXT N
    40 END
9.2 100 INPUT M$
    110 IF M$="N" OR M$="S" OR M$="E" OR M$="W"
        THEN GOTO 500
    120 S=INSTR(M$,"  ")
    130 IF S=0 OR S=1 THEN GOTO 150
    140 IF INSTR(S+1,M$,"  ")=0 THEN GOTO 700
    150 PRINT "I don't understand that":GOTO 100
10.1 10 SCREEN 3:CLS
    20 N=RND(-TIME)
    30 X%=RND(1)*256:Y%=RND(1)*192:C%=1+RND(1)*15
    40 PSET(X%,Y%),C%:PSET(255-X%,Y%),C%
    50 PSET(X%,191-Y%),C%:PSET(255-X%,191-Y%),C%
    60 GOTO 30
10.2 (a) 100 LINE-STEP(20,20),,BF:LINE STEP(-20,-30)-STE
             P(+20,0):LINE STEP(-10,-10)-STEP(0,+20)
         110 RETURN
    (b) 10 SCREEN 2:COLOR 13,15,15:CLS
        20 FOR N=0 TO 4
        30 READ X,Y:PSET(X,Y):GOSUB 100
        40 NEXT N
        50 GOTO 50
        60 DATA 50,50,150,50,50,100,150,100
        100 LINE-STEP(20,20),,BF:LINE STEP(-20,-30)-
            STEP(+20,0)
        110 LINE STEP(-10,-10)-STEP(0,+20)
        120 RETURN
10.3 10 SCREEN 2:COLOR 2,15,15:CLS
    20 LINE(20,20)-(180,180)
    30 LINE(100,30)-(170,170),8:LINE-(30,170),8:LINE-
       (100,30),8
    35 PAINT (100,100),8,8
    40 CIRCLE(100,110),30,4:PAINT(100,110),4,4
    50 GOTO 50
10.4 40 CIRCLE(128,96),50,,,,1/A
10.5 10 SCREEN 2:COLOR 1,15,15:CLS
    20 H$="U20D10R12U10D20BR8":E$="U20NR12D10
       NR10D10R12BR8"
    30 L$="NU20R12BR8":O$="U20R12D20L12"
    40 W$=H$+E$+L$+L$+O$
    50 DRAW"BM80,30;A0;XW$;BM220,30;A3;XW$;"
```

```
        60 DRAW"BM180,150;A2;XW$;BM40,150;A1;XW$;"
        70 GOTO 70
11.1    KEY 1,"GOSUB":KEY 2,"RETURN"
11.2    10   ON STOP GOSUB 500
        20   STOP ON
             .
             .
        500 PRINT"THIS PROGRAM IS BREAK-PROOFED"
        510 PRINT"PRESS ANY KEY TO CONTINUE"
        520 A$=INPUT$(1)
        530 IF A$="S" OR A$="s" THEN STOP
        540 RETURN
12.1    10 P$="O4L8A.L16BL8AGF+GL4A"
        20 PLAY"T100V12;XP$;L8EF+L4GL8F+GL4A;XP$;L4
           EAL8F+L4D."
12.2    10 SOUND 11,220:SOUND 12,5:SOUND 13,11
        20 SOUND 4,243:SOUND 5,0
        30 SOUND 10,16:SOUND 7,&B111011
        40 TIME=0
        50 IF TIME<100 THEN 50
        60 SOUND 10,0
        70 END
13.1    10 FOR I=1 TO 10
        20 INPUT"Number";X(I):NEXT I
        30 PRINT"Set tape to RECORD, then press any key."
           :A$=INPUT$(1)
        40 OPEN"CAS:NUM" FOR OUTPUT AS #1
        50 FOR I=1 TO 10:PRINT #1,X(I):NEXT I
        60 CLOSE #1
        70 PRINT"Now stop the tape.":END
13.2    10 PRINT"Set tape to PLAY."
        20 OPEN"CAS:NUM" FOR INPUT AS #1
        30 FOR I=1 TO 10:INPUT #1,X(I):NEXT I
        40 CLOSE #1
        50 END
13.3    10 SCREEN 2:COLOUR 1, 15, 15:CLS
        20 OPEN"GRP:" FOR OUTPUT AS #1
        30 FOR I=1 TO 14
        40 COLOR I:READ C$
        50 PRESET(20,10*(I+1))
        60 PRINT #1, USING "##";I;:PRINT #1,SPC(10)C$
        70 NEXT I
```

```
80 GOTO 80
90 DATA Black,Medium green,Light green,Dark blue,
   Light blue
100 DATA Dark red,Cyan,Medium red,Light red,Dark yellow
110 DATA Light yellow,Dark green,Magenta,Grey
```

14.1 (a) Text mode 1, 16×16 magnified sprites
(b) Graphics mode 3, 8×8 unmagnified sprites
(c) Graphics mode 2, 8×8 magnified sprites

```
10 SCREEN 2,0:COLOR 8,5,5:CLS
20 SPRITE$(0)=CHR$(0)+CHR$(0)+CHR$(24)+CHR$(60)+
   CHR$(60) +CHR$(24)+CHR$(0)+CHR$(0)
30 SPRITE$(1)=CHR$(24)+CHR$(60)+CHR$(102)+
      CHR$(195)+CHR$(195)+CHR$(102)+CHR$(60)+CHR$(24)
40 PUT SPRITE 0,(50,50),12,0:PUT SPRITE 1,(50,50),8,1
50 GOTO 50
```

15.1

```
10 REM Squashed circle
20 ON ERROR GOTO 110
30 PI=4*ATN(1)
40 SCREEN 2:COLOR 1,15,15:CLS
50 FOR T=0 TO 2*PI STEP 0.01
60 X=128+80*SIN(T):IF X<50 OR X>206 THEN
   ERROR 255
70 Y=96+80*COS(T):IF Y<18 OR Y>174 THEN ERROR 255
80 PSET (X,Y)
90 NEXT T
100 GOTO 100
110 IF ERR=255 THEN 130
120 ON ERROR GOTO 0
130 IF X<50 THEN X=50:RESUME 70
140 IF X>206 THEN X=206:RESUME 70
150 IF Y<18 THEN Y=18:RESUME 80
160 Y=174:RESUME 80
```

16.1

```
10 CLEAR 255,&HF370
20 FOR X=0 TO 3
30 READ H$
40 POKE &HF370+C,VAL("&H"+H$):NEXT C
50 DATA CD,C0,00,C9
60 DEF USR0=&HF370
70 X=USR0(0)
80 END
```

17.1

```
10 SCREEN 1
20 S=BASE(7)
```

```
30 FOR C=0 TO 31
40 READ X:VPOKE S+C,X:NEXT C
50 DATA 1,3,7,15,31,63,127,255
60 DATA 128,192,224,240,248,252,254,255
70 DATA 255,254,252,248,240,224,192,128
80 DATA 255,127,63,31,15,7,3,1
90 END
```

17.2

```
10 SCREEN 1
20 CLEAR 255,&HEB80

.

999 REM Save characters
1000 FOR I=0 TO &H800
1010 POKE &HEB80+I,VPEEK(BASE(7)+I):NEXT I
1020 PRINT"Set tape to RECORD then press any key":
     A$=INPUT$(1)
1030 BSAVE"CAS:CHAR",&HEB80,&HF37F
1040 END

1999 REM Load characters
2000 SCREEN 1:CLEAR 255,&HEB80
2010 PRINT"Set tape to PLAY."
2020 BLOAD"CAS:CHAR"
2030 FOR I=0 TO &H800
2040 VPOKE BASE(7)+I,PEEK(&HEB80+I):NEXT I
2050 END
```

Appendix A
Characters and ASCII Codes

Code	*Character*	*Code*	*Character*	*Code*	*Character*	*Code*	*Character*
0		64	@	128	ç	192	
1	☺	65	A	129	ü	193	
2	⊕	66	B	130	ē	194	
3	♥	67	C	131	å	195	
4	♦	68	D	132	ã	196	
5	♤	69	E	133	à	197	
6	♠	70	F	134	å	198	
7	-	71	G	135	ç	199	
8	◘	72	H	136	ê	200	
9	○	73	I	137	ë	201	
10	◙	74	J	138	è	202	
11	♂	75	K	139	ï	203	
12	♀	76	L	140	î	204	
13	♪	77	M	141	ì	205	▼
14	♫	78	N	142	Ä	206	▲
15	*	79	O	143	Å	207	►
16	†	80	P	144	È	208	◄
17	⊥	81	Q	145	æ	209	↕
18	⊤	82	R	146	Æ	210	‼
19	⊣	83	S	147	ô	211	
20	⊢	84	T	148	ö	212	
21	+	85	U	149	ò	213	
22	|	86	V	150	û	214	
23	—	87	W	151	ù	215	
24	⌐	88	X	152	ÿ	216	△
25	¬	89	Y	153	Ö	217	‡
26	⌙	90	Z	154	Ü	218	ω
27	⌟	91	[	155	¢	219	
28	×	92	·.	156	£	220	
29	/	93	]	157	¥	221	
30	\	94	^	158	Pt	222	

31	←	95	_	159	ƒ	223	▀
32		96	`	160	á	224	α
33	!	97	a	161	í	225	β
34	"	98	b	162	ó	226	Γ
35	#	99	c	163	ú	227	π
36	$	100	d	164	ñ	228	Σ
37	%	101	e	165	Ñ	229	σ
38	&	102	f	166	ª	230	μ
39	'	103	g	167	º	231	τ
40	(	104	h	168	¿	232	Φ
41	)	105	i	169	⌐	233	θ
42	*	106	j	170	¬	234	Ω
43	+	107	k	171	½	235	δ
44	,	108	l	172	¼	236	∞
45	-	109	m	173	¡	237	φ
46	.	110	n	174	«	238	ε
47	/	111	o	175	»	239	∩
48	0	112	p	176	Ã	240	≡
49	1	113	q	177	ã	241	±
50	2	114	r	178	Ĩ	242	≥
51	3	115	s	179	ĩ	243	≤
52	4	116	t	180	Õ	244	⌠
53	5	117	u	181	õ	245	⌡
54	6	118	v	182	Ũ	246	÷
55	7	119	w	183	ũ	247	≈
56	8	120	x	184	Ij	248	°
57	9	121	y	185	ij	249	∙
58	:	122	z	186	¾	250	·
59	;	123	{	187	~	251	√
60	<	124	¦	188	◇	252	η
61	=	125	}	189	⅞	253	2
62	>	126	~	190	π	254	▌
63	?	127	Δ	191	§	255	■

Appendix B
Error Messages

Code	*Message*	*Description*
1	NEXT without FOR	A NEXT statement has no corresponding FOR statement, usually because different variables have been used in the FOR and NEXT statements or because a GOTO statement goes to the middle of the FOR...NEXT loop.
2	Syntax error	A statement contains a typing error or is wrongly punctated.
3	RETURN GOSUB without	A RETURN statement has no corresponding GOSUB statement, usually because a GOTO statement transfers execution to the subroutine, or because the main program routine does not end with a GOTO or END statement.
4	Out of DATA	There is no more data to be read by a READ statement. A data item may have been omitted, or a RESTORE statement may be needed.
5	Illegal function call	The argument of a function is not within the required range.
6	Overflow	A number is too large for the computer to handle, or an address parameter is not within the required range.
7	Out of memory	The program is too long, or contains too many loops or subroutines, or too many variables have been used.
8	Undefined line number	A non-existent line number has been used in a GOTO, GOSUB or similar statement.
9	Subscript out of range	The subscript of an array element exceeds the number specified in a DIM statement, or exceeds 11 if the array has not been DIMensioned.
10	Redimensioned array	The same name has been used for two different arrays, or an ERASE statement has been omitted.

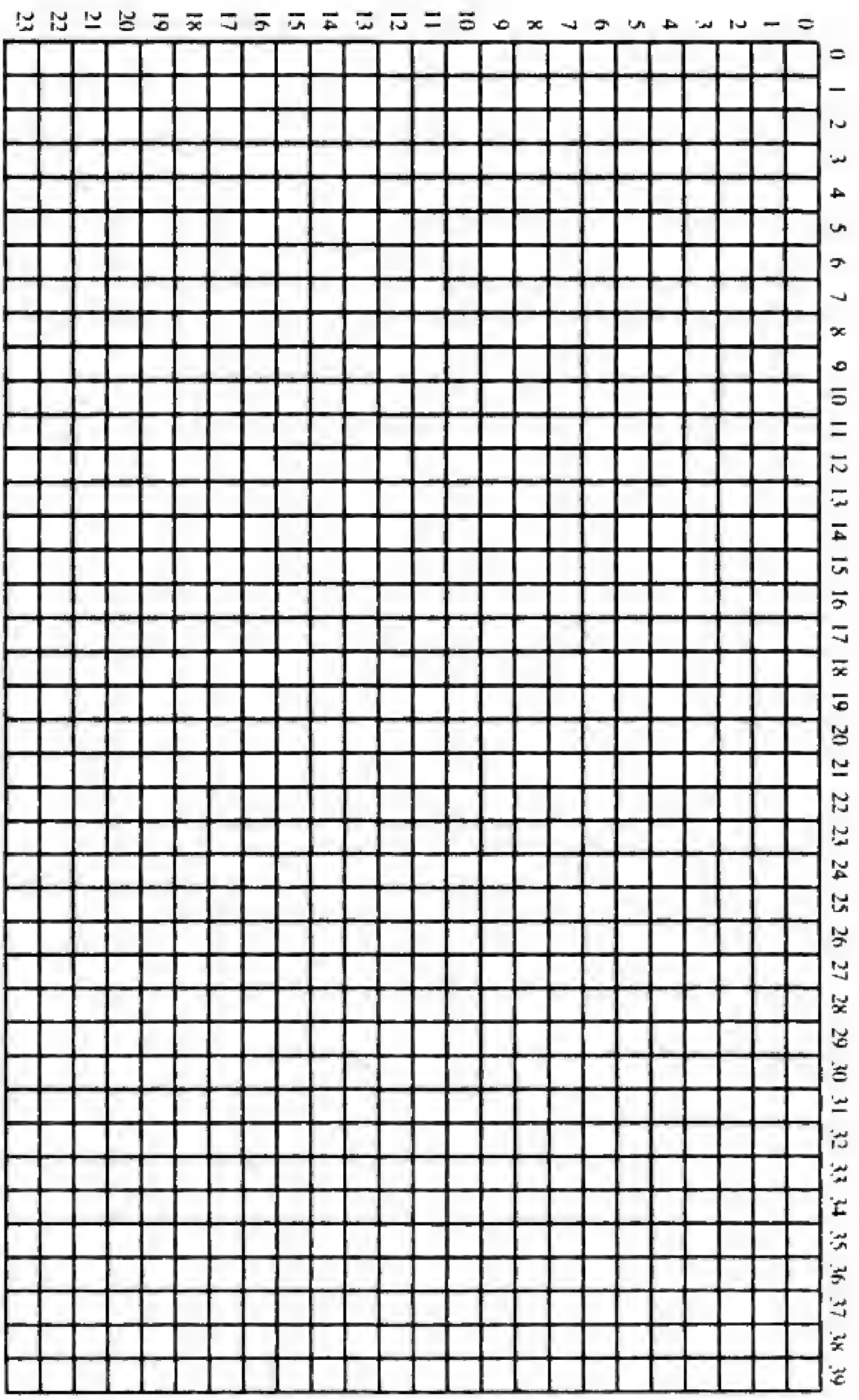
11	Division by zero	The evaluation of an expression involves dividing by zero, or by a variable which has not been assigned a value.
12	Illegal direct	An attempt has been made to use a statement which can only be used in a program, such as a DEFFN statement, in direct mode.
13	Type mismatch	A string has been assigned to a numeric variable or vice versa, or the wrong data type has been used with a function or operator.
14	Out of string space	The computer has run out of string storage space, or the amount of string space specified in a CLEAR statement is too small.
15	String too long	An attempt has been made to create a string containing more than 255 characters.
16	String formula too complex	A string expression is too complex for the computer to handle.
17	Can't CONTINUE	A CONT statement has been used within a program, or an attempt has been made to CONTinue a program which ended with an error or has been edited.
18	Undefined user function	An attempt has been made to use an FN function which has not been defined by a DEF FN statement.
19	Device I/O error	The execution of an input or output command has been halted by an error or by pressing <CTRL> and <STOP>.
20	Verify error	An attempt to verify a recording using CLOAD? has failed, because the program on tape differs from the program in memory.
21	No RESUME	An error-processing routine does not end with RESUME, END or ON ERROR GOTO 0.
22	RESUME without error	A RESUME statement has been encountered when no error has occurred, because a GOTO statement transfers execution to the error-trapping routine or because the main program routine does not end with a GOTO or END statement.
23	Unprintable error	There is an error, but no error message.
24	Missing operand	An expression contains an operator with no operand, or insufficient operands.
25	Line buffer overflow	The input line buffer is full, i.e. the line being entered contains too many characters.

26–49	Unprintable errors	These error numbers are reserved for future expansion.
50	Field overflow	
51	Internal error	The fault lies in the computer, not in your program.
52	Bad file number	A file number exceeds the range specified by MAXFILES statement, or an attempt has been made to use a file which has not been OPENed.
53	File not found	
54	File already open	An attempt has been made to open a file which is already open, or the same file number has been used for two different files.
55	Input past end	An INPUT# command has been used when there is no more data in the file.
56	Bad file name	A file has been given an unacceptable name, or the device name in an OPEN, SAVE or LOAD statement is unacceptable.
57	Direct statement in file	The file being loaded by a LOAD command contains a statement without a line number, or is a data file, not a program file.
58	Sequential I/O only	
59	File not open	An attempt has been made to use a file which has not been OPENed.

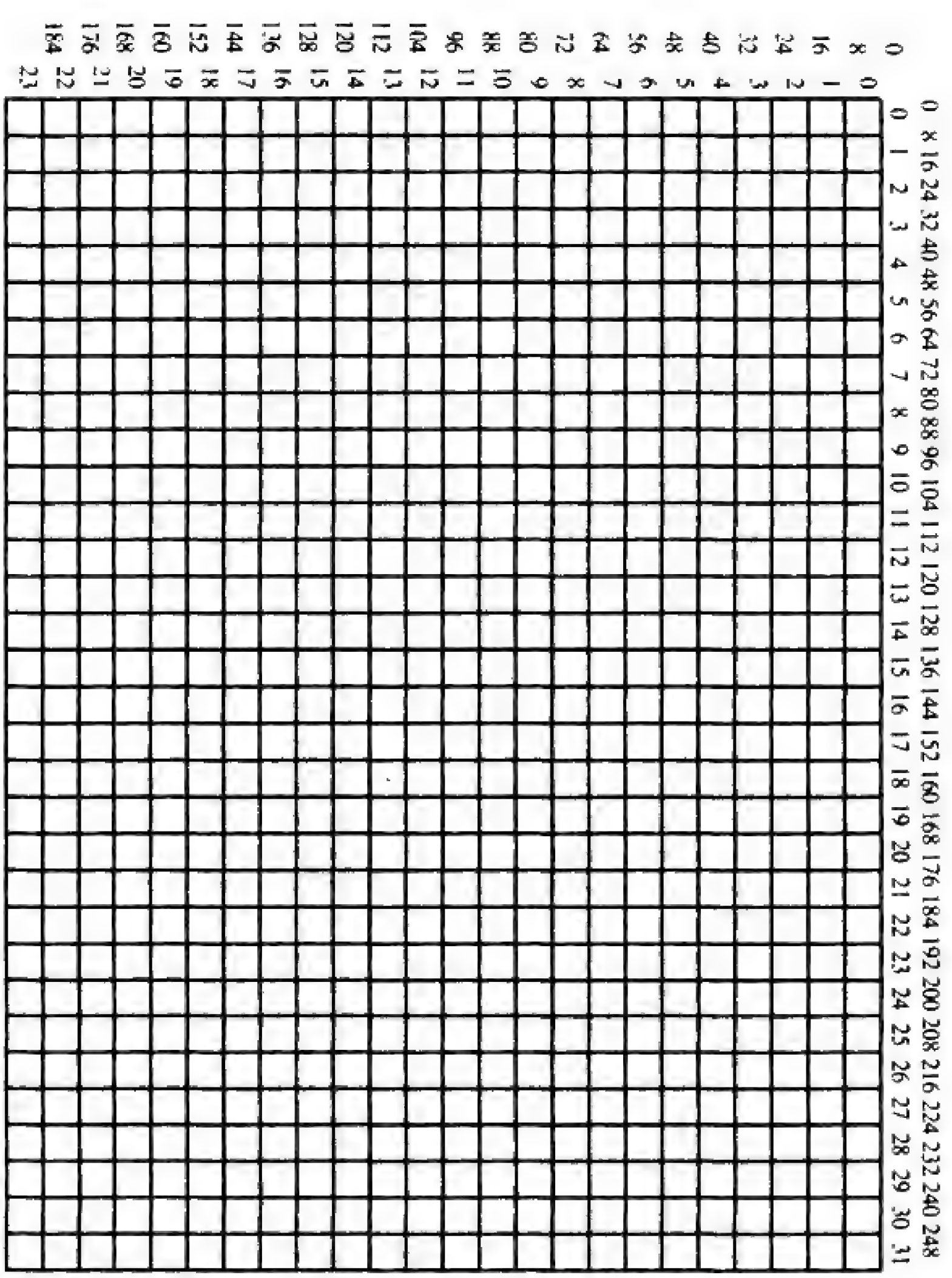
60–255 Spare codes, available for user-defined errors.

Appendix C
Sample Graphics Design Sheets

Mode 0



Modes 1, 2 and 3



Appendix D
Decimal/Hex Conversion Table

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
20	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
30	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
40	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
50	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
60	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
70	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
90	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A0	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B0	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C0	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D0	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E0	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F0	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

How to Use the Table

To convert a decimal number between 0 and 255 to hex:

Find the number in the table. The number at the left of the row will give the first hex digit; the number at the top of the column will give the second digit.

To convert a two-digit hex number to decimal:
Find the first digit in the left-hand column. This determines which row of the table the answer is in. Now find the second digit in the top row. This determines which column the answer is in. The answer will be found where the row and column intersect.

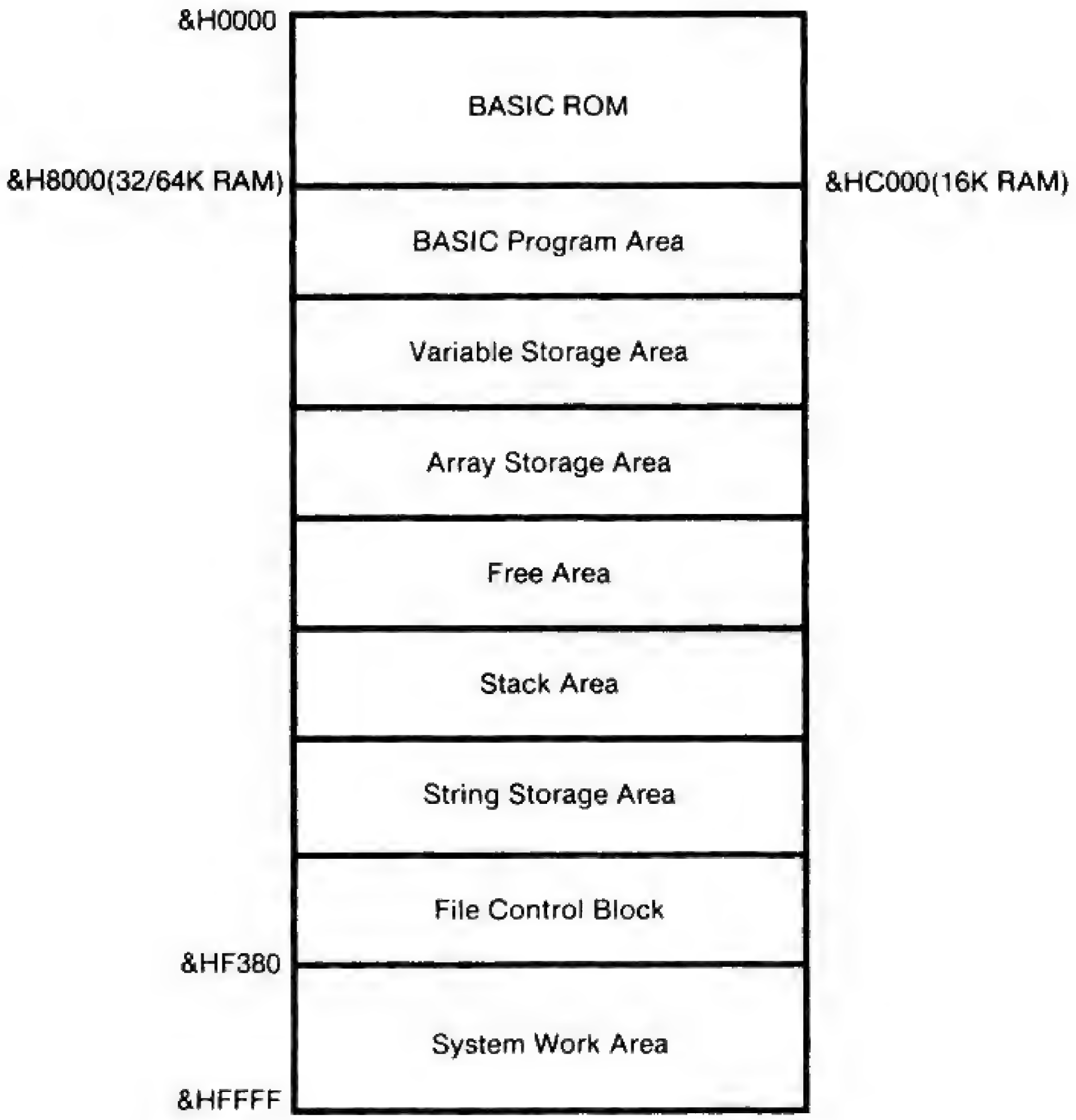
Appendix E
Control Code and <CTRL> Key Functions

Control code	*Control key*	*Special key*	*Function*
02	<CTRL>+<B>		moves the cursor left to the beginning of a word
03	<CTRL>+<C>		releases input wait state, cancels automatic line numbering
05	<CTRL>+<E>		deletes the character under the cursor and any characters to the right of it
06	<CTRL>+<F>		moves cursor right to the beginning of a word
07	<CTRL>+<G>		beep
08	<CTRL>+<H>	<BS>	backspaces
09	<CTRL>+<I>	<TAB>	moves cursor to the next TAB position
10	<CTRL>+<J>		moves cursor to the start of the next line (line feed)
11	<CTRL>+<K>	<HOME>	moves cursor to the top left-hand corner of the screen
12	<CTRL>+<L>	<CLS>	clears the screen and moves the cursor to the top left-hand corner
13	<CTRL>+<M>	<RETURN>	carriage return (enters the current line)
14	<CTRL>+<N>		moves the cursor to the end of the line
18	<CTRL>+<R>	<INS>	switches on insert mode
21	<CTRL>+<U>		deletes the current line
24	<CTRL>+<X>	<SELECT>	—
27	<CTRL>+<[>	<ESC>	—
28	<CTRL>+< \ >		moves the cursor one column right
29	<CTRL>+<]>		moves the cursor one column left
30	<CTRL>+<^>		moves the cursor one row up
31	<CTRL>+<->		moves the cursor one row down

Appendix F
Colour Codes

Code	*Colour*
0	transparent
1	black
2	medium green
3	light green
4	dark blue
5	light blue
6	dark red
7	cyan/sky blue
8	medium red
9	light red
10	dark yellow
11	light yellow
12	dark green
13	magenta
14	grey
15	white

Appendix G
Memory Map



BASIC ROM

This area contains the BASIC interpreter.

BASIC Program Area

Your BASIC programs will be stored here.

Variable Storage Area

This area stores numeric variables and, for string variables, pointers to the string storage area. The VARPTR function can be used to find the address for a particular variable.

Array Storage Area

This area stores numeric array variables and pointers to the string storage area for string array variables. Its size is increased when a DIM statement is executed.

Free Area

The unused area, the size of which can be found by using the FRE function.

Stack Area

Return addresses, for loops and GOSUB statements, are stored here.

String Storage Area

String variables are stored here. The size of the area can be specified in a CLEAR statement.

File Control Block

This area is used for input and output operations using files. Its size can be altered by using a MAXFILES statement. Space can be created for machine-code routines by moving the upper limit of the file control block down with a CLEAR statement.

System Work Area

This area is used by the BASIC interpreter. System variables are stored here.

Appendix H
Z80 Opcodes

ADC A, (HL)	8E	ADC A, (IX+d)	DD8Ed
ADC A, (IY+d)	FD8Ed	ADC A,A	8f
ADC A,B	88	ADC A,C	89
ADC A,D	8A	ADC A,E	8B
ADC A,H	8C	ADC A,L	8D
ADC A,n	CEn	ADC HL,BC	ED4A
ADC HL,DE	ED5A	ADC HL,HL	ED6A
ADC HL,SP	ED7A	ADD A, (HL)	86
ADD A, (IX+d)	DD86d	ADD A, (IY+d)	FD86d
ADD A,A	87	ADD A,B	80
ADD A,C	81	ADD A,D	82
ADD A,E	83	ADD A,H	84
ADD A,L	85	ADD A,n	C6n
ADD HL,BC	09	ADD HL,DE	19
ADD HL,HL	29	ADD HL,SP	39
ADD IX,BC	DD09	ADD IX,DE	DD19
ADD IX,IX	DD29	ADD IX,SP	DD39
ADD IY,BC	FD09	ADD IY,DE	FD19
ADD IY,IY	FD29	ADD IY,SP	FD39
AND (HL)	A6	AND (IX+d)	DDA6d
AND (IY+d)	FDA6d	AND A	A7
AND B	A0	AND C	A1
AND D	A2	AND E	A3
AND H	A4	AND L	A5
AND n	E6n	BIT 0, (HL)	CB46
BIT 0, (IX+d)	DDCBd46	BIT 0, (IY+d)	FDCBd46
BIT 0,A	CB47	BIT 0,B	CB40
BIT 0,C	CB41	BIT 0,D	CB42
BIT 0,E	CB43	BIT 0,H	CB44
BIT 0,L	CB45	BIT 1,(HL)	CB4E
BIT 1, (IX+d)	DDCBd4E	BIT 1, (IY+d)	FDCBd4E
BIT 1,A	CB4F	BIT 1,B	CB48

BIT 1,C	CB49	BIT 1,D	CB4A
BIT 1,E	CB4B	BIT 1,H	CB4C
BIT 1,L	CB4D	BIT 2, (HL)	CB56
BIT 2, (IX+d)	DDCBd56	BIT 2, (IY+d)	FDCBd56
BIT 2,A	CB57	BIT 2,B	CB50
BIT 2,C	CB51	BIT 2,D	CB52
BIT 2,E	CB53	BIT 2,H	CB54
BIT 2,L	CB55	BIT 3, (HL)	CB5E
BIT 3, (IX+d)	DDCBd5E	BIT 3, (IY+d)	FDCBd5E
BIT 3,A	CB5F	BIT 3,B	CB58
BIT 3,C	CB59	BIT 3,D	CB5A
BIT 3,E	CB5B	BIT 3,H	CB5C
BIT 3,L	CB5D	BIT 4, (HL)	CB66
BIT 4, (IX+d)	DDCBd66	BIT 4, (IY+d)	FDCBd66
BIT 4,A	CB67	BIT 4,B	CB60
BIT 4,C	CB61	BIT 4,D	CB62
BIT 4,E	CB63	BIT 4,H	CB64
BIT 4,L	CB65	BIT 5, (HL)	CB6E
BIT 5, (IX+d)	DDCBd6E	BIT 5, (IY+d)	FDCBd6E
BIT 5,A	CB6F	BIT 5,B	CB68
BIT 5,C	CB69	BIT 5,D	CB6A
BIT 5,E	CB6B	BIT 5,H	CB6C
BIT 5,L	CB6D	BIT 6, (HL)	CB76
BIT 6, (IX+d)	DDCBd76	BIT 6, (IY+d)	FDCBd76
BIT 6,A	CB77	BIT 6,B	CB70
BIT 6,C	CB71	BIT 6,D	CB72
BIT 6,E	CB73	BIT 6,H	CB74
BIT 6,L	CB75	BIT 7, (HL)	CB7E
BIT 7, (IX+d)	DDCBd7E	BIT 7,(IY+d)	FDCBd7E
BIT 7,A	CB7F	BIT 7,B	CB78
BIT 7,C	CB79	BIT 7,D	CB7A
BIT 7,E	CB7B	BIT 7,H	CB7C
BIT 7,L	CB7D	CALL C,nn	DCnn
CALL M,nn	FCnn	CALL NC,nn	D4nn
CALL nn	CDnn	CALL NZ,nn	C4nn
CALL P,nn	F4nn	CALL PE,nn	ECnn
CALL PO,nn	E4nn	CALL Z,nn	CCnn
CCF	3F	CP (HL)	BE
CP (IX+d)	DDBEd	CP (IY+d)	FDBEd
CP A	BF	CP B	B8
CP C	B9	CP D	BA
CP E	BB	CP H	BC

CP L	BD	CP n	FEn
CPD	EDA9	CPDR	EDB9
CPI	EDA1	CPIR	EDB1
CPL	2F	DAA	27
DEC (HL)	35	DEC (IX+d)	DD35d
DEC (IY+d)	FD35d	DEC A	3D
DEC B	05	DEC BC	0B
DEC C	0D	DEC D	15
DEC DE	1B	DEC E	1D
DEC H	25	DEC HL	2B
DEC IX	DD2B	DEC IY	FD2B
DEC L	2D	DEC SP	3B
DI	F3	DJNZ,d	10d
EI	FB	EX (SP),HL	E3
EX (SP) ,IX	DDE3	EX (SP) ,IY	FDE3
EX AF, AF	08	EX DE,HL	EB
EXX	D9	HALT	76
IM 0	ED46	IM 1	ED56
IM 2	ED5E	IN A, (C)	ED78
IN A, (n)	DBn	IN B, (C)	ED40
IN C, (C)	ED48	IN D, (C)	ED50
IN E, (C)	ED58	IN H, (C)	ED60
IN L, (C)	ED68	INC (HL)	34
INC (IX+d)	DD34d	INC (IY+d)	FD34d
INC A	3C	INC B	04
INC BC	03	INC C	0C
INC D	14	INC DE	13
INC E	1C	INC H	24
INC HL	23	INC IX	DD23
INC IY	FD23	INC L	2C
INC SP	33	IND	EDAA
INDR	EDBA	INI	EDA2
INIR	EDB2	JP (HL)	E9
JP (IX)	DDE9	JP (IY)	FDE9
JP C,nn	DAnn	JP M,nn	FAnn
JP NC,nn	D2nn	JP nn	C3nn
JP NZ,nn	C2nn	JP P,nn	F2nn
JP PE,nn	EAnn	JP PO,nn	E2nn
JP Z,nn	CAnn	JR C,d	38d
JR ,d	18d	JR NC,d	30d
JR NZ,d	20d	JR Z,d	28d
LD (BC),A	02	LD (DE) ,A	12

LD (HL) ,A	77	LD (HL) ,B	70
LD (HL) ,C	71	LD (HL) ,D	72
LD (HL) ,E	73	LD (HL),H	74
LD (HL) ,L	75	LD (HL),n	36n
LD (IX+d) ,A	DD77d	LD (IX+d),B	DD70d
LD (IX+d) ,C	DD71d	LD (IX+d) ,D	DD72d
LD (IX+d) ,E	DD73d	LD (IX+d) ,H	DD74d
LD (IX+d),L	DD75d	LD (IX+d) ,n	DD36dn
LD (IY+d) ,A	FD77d	LD (IY+d) ,B	FD70d
LD (IY+d) ,C	FD71d	LD (IY+d) ,D	FD72d
LD (IY+d) ,E	FD73d	LD (IY+d) ,H	FD74d
LD (IY+d) ,L	FD75d	LD (IY+d) ,n	FD36dn
LD (nn) ,A	32nn	LD (nn) ,BC	ED43nn
LD (nn) ,DE	ED53nn	LD (nn) ,HL	22nn
LD (nn) ,IX	DD22nn	LD (nn) ,IY	FD22nn
LD (nn) ,SP	ED73nn	LD A, (BC)	0A
LD A, (DE)	1A	LD A, (HL)	7E
LD A, (IX+d)	DD7E	LD A, (IX+d)	FD7Ed
LD A, (nn)	3Ann	LD A,A	7F
LD A,B	78	LD A,C	79
LD A,D	7A	LD A,E	7B
LD A,H	7C	LD A,I	ED57
LD A,L	7D	LD A,n	3En
LD B, (HL)	46	LD B, (IX+d)	DD46d
LD B, (IY+d)	FD46d	LD B,A	47
LD B,B	40	LD B,C	41
LD B,D	42	LD B,E	43
LD B,H	44	LD B,L	45
LD B,n	06n	LD BC, (nn)	ED4Bnn
LD BC,nn	01nn	LD C, (HL)	4E
LD C, (IX+d)	DD4Ed	LD C, (IY+d)	FD4Ed
LD C,A	4F	LD C,B	48
LD C,C	49	LD C,D	4A
LD C,E	4B	LD C,H	4C
LD C,L	4D	LD C,n	0En
LD D, (HL)	56	LD D, (IX+d)	DD56d
LD D, (IY+d)	FD56d	LD D,A	57
LD D,B	50	LD D,C	51
LD D,D	52	LD D,E	53
LD D,H	54	LD D,L	55
LD D,n	16n	LD DE, (nn)	ED58nn
LD DE,nn	11nn	LD E,(HL)	5E

LD E, (IX+d)	DD5Ed	LD E, (IY+d)	FD5Ed
LD E,A	5F	LD E,B	58
LD E,C	59	LD E,D	5A
LD E,E	5B	LD E,H	5C
LD E,L	5D	LD E,n	1En
LD H, (HL)	66	LD H, (IX+d)	DD66d
LD H, (IY+d)	FD66d	LD H,A	67
LD H,B	60	LD H,C	61
LD H,D	62	LD H,E	63
LD H,H	64	LD H,L	65
LD H,n	26n	LD I,A	ED47
LD IX, (nn)	DD2Ann	LD IX,nn	DD21nn
LD IY, (nn)	FD2Ann	LD IY,nn	FD21nn
LD L, (HL)	6E	LD L, (IX+d)	DD6Ed
LD L, (IY+d)	FD6Ed	LD L,A	6F
LD L,B	68	LD L,C	69
LD L,D	6A	LD L,E	6B
LD L,H	6C	LD L,L	6D
LD L,n	2En	LD SP, (nn)	ED7Bnn
LD SP,HL	F9	LD SP,IX	DDF9
LD SP,IY	FDF9	LD SP,nn	31nn
LDD	EDA8	LDDR	EDB8
LDI	EDA0	LDIR	EDB0
NEG	ED44	NOP	00
OR (HL)	B6	OR (IX+d)	DDB6d
OR (IY+d)	FDB6d	OR A	B7
OR B	B0	OR C	B1
OR D	B2	OR E	B3
OR H	B4	OR L	B5
OR n	F6n	OTDR	EDBB
OTIR	EDB3	OUT (C) ,A	ED79
OUT (C) ,B	ED41	OUT (C) ,C	ED49
OUT (C) ,D	ED51	OUT (C) ,E	ED59
OUT (C) ,H	ED61	OUT (C) ,L	ED69
OUT (n) ,A	D3n	OUTD	EDAB
OUTI	EDA3	POP AF	F1
POP BC	C1	POP DE	D1
POP HL	E1	POP IX	DDE1
POP IY	FDE1	PUSH AF	F5
PUSH BC	C5	PUSH DE	D5
PUSH HL	E5	PUSH IX	DDE5
PUSH IY	FDE5	RES 0, (HL)	CB86

RES 0, (IX+d)	DDCBd86	RES 0, (IY+d)	FDCBd86
RES 0,A	CB87	RES 0,B	CB80
RES 0,C	CB81	RES 0,D	CB82
RES 0,E	CB83	RES 0,H	CB84
RES 0,L	CB85	RES 1, (HL)	CB8E
RES 1, (IX+d)	DDCBd8e	RES 1, (IY+d)	FDCBd8E
RES 1,A	CB8F	RES 1,B	CB88
RES 1,C	CB89	RES 1,D	CB8A
RES 1,E	CB8B	RES 1,H	CB8C
RES 1,L	CB8D	RES 2, (HL)	CB96
RES 2, (IX+d)	DDCBd96	RES 2, (IY+d)	FDCBd96
RES 2,A	CB97	RES 2,B	CB90
RES 2,C	CB91	RES 2,D	CB92
RES 2,E	CB93	RES 2,H	CB94
RES 2,L	CB95	RES 3, (HL)	CB9E
RES 3, (IX+d)	DDCBd9E	RES 3, (IY+d)	FDCBd9E
RES 3,A	CB9F	RES 3,B	CB98
RES 3,C	CB99	RES 3,D	CB9A
RES 3,E	CB9B	RES 3,H	CB9C
RES 3,L	CB9D	RES 4, (HL)	CBA6
RES 4, (IX+d)	DDCBdA6	RES 4, (IY+d)	FDCBdA6
RES 4 ,A	CBA7	RES 4,B	CBA0
RES 4,C	CBA1	RES 4,D	CBA2
RES 4,E	CBA3	RES 4,H	CBA4
RES 4,L	CBA5	RES 5, (HL)	CBAE
RES 5, (IX+d)	DDCBdAE	RES 5, (IY+d)	FDCBdAE
RES 5,A	CBAF	RES 5,B	CBA8
RES 5,C	CBA9	RES 5,D	CBAA
RES 5,E	CBAB	RES 5,H	CBAC
RES 5,L	CBAD	RES 6, (HL)	CBB6
RES 6, (IX+d)	DDCBdB6	RES 6, (IY+d)	FDCBdB6
RES 6,A	CBB7	RES 6,B	CBB0
RES 6,C	CBB1	RES 6,D	CBB2
RES 6,E	CBB3	RES 6,H	CBB4
RES 6,L	CBB5	RES 7,(HL)	CBBE
RES 7, (IX+d)	DDCBdBE	RES 7, (IY+d)	FDCBdBE
RES 7,A	CBBF	RES 7,B	CBB8
RES 7,C	CBB9	RES 7,D	CBBA
RES 7,E	CBBB	RES 7,H	CBBC
RES 7,L	CBBD	RET	C9
RET C	D8	RET M	F8
RET NC	D0	RET NZ	C0

RET P	F0	RET PE	E8
RET PO	E0	RET Z	C8
RETI	ED4D	RETN	ED45
RL (HL)	CB16	RL (IX+d)	DDCBd16
RL (IY+d)	FDCBd16	RL A	CB17
RL B	CB10	RL C	CB11
RL D	CB12	RL E	CB13
RL H	CB14	RL L	CB15
RLA	17	RLC (HL)	CB06
RLC (IX+d)	DDCBd06	RLC (IY+d)	FDCBd06
RLC A	CB07	RLC B	CB00
RLC C	CB01	RLC D	CB02
RLC E	CB03	RLC H	CB04
RLC L	CB05	RLCA	07
RLD	ED6F	RR (HL)	CB1E
RR (IX+d)	DDC8d1E	RR (IY+d)	FDC8d1E
RR A	CB1F	RR B	CB18
RR C	CB19	RR D	CB1A
RR E	CB1B	RR H	CB1C
RR L	CB1D	RRA	1F
RRC (HL)	CB0E	RRC (IX+d)	DDCBd0E
RRC (IY+d)	FDCBd0E	RRC A	CB0F
RRC B	CB08	RRC C	CB09
RRC D	CB0A	RRC E	CB0B
RRC H	CB0C	RRC L	CB0D
RRCA	0F	RRD	ED67
RST 0	C7	RST 10H	D7
RST 18H	DF	RST 20H	E7
RST 28H	EF	RST 30H	F7
RST 38H	FF	RST 8	CF
SBC A, (HL)	9E	SBC A, (IX+d)	DD9Ed
SBC A, (IY+d)	FD9Ed	SBC A,A	9F
SBC A,B	98	SBC A,C	99
SBC A,D	9A	SBC A,E	9B
SBC A,H	9C	SBC A,L	9D
SBC A,n	DEn	SBC HL,BC	ED42
SBC HL,DE	ED52	SBC HL,HL	ED62
SBC HL,SP	ED72	SCF	37
SET 0, (HL)	CBC6	SET 0, (IX+d)	DDCBdC6
SET 0, (IY+d)	FDCBdC6	SET 0,A	CBC7
SET 0,B	CBC0	SET 0,C	CBC1
SET 0,D	CBC2	SET 0,E	CBC3

SET 0,H	CBC4	SET 0,L	CBC5
SET 1, (HL)	CBCE	SET 1, (IX+d)	DDCBdCE
SET 1, (IY+d)	FDCBdCE	SET 1,A	CBCF
SET 1,B	CBC8	SET 1,C	CBC9
SET 1,D	CBCA	SET 1,E	CBCB
SET 1,H	CBCC	SET 1,L	CBCD
SET 2, (HL)	CBD6	SET 2, (IX+d)	DDCBdD6
SET 2, (IY+d)	FDCBdD6	SET 2,A	CBD7
SET 2,B	CBD0	SET 2,C	CBD1
SET 2,D	CBD2	SET 2,E	CBD3
SET 2,H	CBD4	SET 2,L	CBD5
SET 3, (HL)	CBDE	SET 3, (IX+d)	DDCBdDE
SET 3, (IY+d)	FDCBdDE	SET 3,A	CBDF
SET 3,B	CBD8	SET 3,C	CBD9
SET 3,D	CBDA	SET 3,E	CBDB
SET 3,H	CBDC	SET 3,L	CBDD
SET 4, (HL)	CBE6	SET 4, (IX+d)	DDCBdE6
SET 4, (IY+d)	FDCBdE6	SET 4,A	CBE7
SET 4,B	CBE0	SET 4,C	CBE1
SET 4,D	CBE2	SET 4,E	CBE3
SET 4,H	CBE4	SET 4,L	CBE5
SET 5, (HL)	CBEE	SET 5, (IX+d)	DDCBdEE
SET 5, (IY+d)	FDCBdEE	SET 5,A	CBEF
SET 5,B	CBE8	SET 5,C	CBE9
SET 5,D	CBEA	SET 5,E	CBEB
SET 5,H	CBEC	SET 5,L	CBED
SET 6, (HL)	CBF6	SET 6, (IX+d)	DDCBdF6
SET 6, (IY+d)	FDCBdF6	SET 6,A	CBF7
SET 6,B	CBF0	SET 6,C	CBF1
SET 6,D	CBF2	SET 6,E	CBF3
SET 6,H	CBF4	SET 6,L	CBF5
SET 7, (HL)	CBFE	SET 7, (IX+d)	DDCBFE
SET 7, (IY+d)	FDCBdFE	SET 7,A	CBFF
SET 7,B	CBF8	SET 7,C	CBF9
SET 7,D	CBFA	SET 7,E	CBFB
SET 7,H	CBFC	SET 7,L	CBFD
SLA (HL)	CB26	SLA (IX+d)	DDCBd26
SLA (IY+d)	FDCBd26	SLA A	CB27
SLA B	CB20	SLA C	CB21
SLA D	CB22	SLA E	CB23
SLA H	CB24	SLA L	CB25
SRA (HL)	CB2E	SRA (IX+d)	DDCBd2E

SRA (IY+d)	FDCBd2E	SRA A	CB2F
SRA B	CB28	SRA C	CB29
SRA D	CB2A	SRA E	CB2B
SRA H	CB2C	SRA L	CB2D
SRL (HL)	CB3E	SRL (IX+d)	DDCBd3E
SRL (IY+d)	FDCBd3E	SRL A	CB3F
SRL B	CB38	SRL C	CB39
SRL D	CB3A	SRL E	CB3B
SRL H	CB3C	SRL L	CB3D
SUB (HL)	96	SUB (IX+d)	DD96d
SUB (IY+d)	FD96d	SUB A	97
SUB B	90	SUB C	91
SUB D	92	SUB E	93
SUB H	94	SUB L	95
SUB n	D6n	XOR (HL)	AE
XOR (IX+d)	DDAEd	XOR (IY+d)	FDAEd
XOR A	AF	XOR B	A8
XOR C	A9	XOR D	AA
XOR E	AB	XOR H	AC
XOR L	AD	XOR n	EEn

Index